Compiling to categories (extended version)

CONAL ELLIOTT, Target

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.

1 INTRODUCTION

As discovered by Joachim Lambek (1980, 1986), the models of the simply typed λ-calculus (STLC) are exactly the cartesian closed categories (CCCs). Moreover, there is a simple, compositional, syntactic transformation from STLC to the vocabulary of CCCs. Each CCC, i.e., each interpretation of this vocabulary, thus gives an interpretation of the STLC. This paper explores a practical application of Lambek's discovery for giving a variety of principled non-standard interpretations of Haskell programs by means of a fairly simple GHC plugin that performs the needed source-to-source transformation to generalize Haskell code to categories other than the usual one. The interpretations we show include compiling Haskell to massively parallel hardware, linear maps (generalized "matrices"), automatic differentiation, incremental computation, and interval analysis, as well as a textual presentation of CCC expressions for debugging. Moreover, these CCCs combine easily and usefully. For instance, one can describe a function directly in Haskell and then apply a few interpretations to compute the exact *n*th derivative (as a linear map), incrementally, and in hardware. As we'll see, we can sometimes get away with less power, particularly with a cartesian category (without closure), although at the cost of larger categorical translations. We can also make use of additional power, particularly bi-cartesian (coproducts), allowing translation of sum types and **case** expressions.

The GHC plugin that implements categorical interpretation is modular in that it has no knowledge of specific target categories. To introduce a new interpretation, one simply defines a type and few Haskell class instances for it—all in regular Haskell code without no exposure to compiler internals. Each interpretation corresponds to a (possibly closed) cartesian functor, and this property makes for a simple specification, useful in calculating the needed class instances.

2 CARTESIAN CLOSED CATEGORIES

There are many introductions to category theory, e.g., (Awodey 2006; Lawvere and Schanuel 2009). For the purposes of this paper, a brief description of interfaces will do. The basic category interface, along with its instance for functions, is as follows:

```
infixr 9 ∘
class Category k where                               instance Category (→) where
  id  :: a ‘k‘ a                      -- identity      id    = λx → x
  (∘) :: (b ‘k‘ c) → (a ‘k‘ b) → (a ‘k‘ c)  -- composition   g ∘ f = λx → g (f x)
```

The category laws state that *id* is indeed the identity for composition and that composition is associative.

A *cartesian* category adds products, with one introduction and two elimination operations:

```
infixr 3 △
class Category k ⇒ Cartesian k where
  (△) :: (a ‘k‘ c) → (a ‘k‘ d) → (a ‘k‘ (c × d))    -- “fork”
  exl :: (a × b) ‘k‘ a    -- extract left
  exr :: (a × b) ‘k‘ b    -- extract right
```

```
instance Cartesian (→) where
  f △ g = λx → (f x, g x)
  exl   = λ(a, b) → a
  exr   = λ(a, b) → b
```

In this paper, “$a \times b$” is a synonym for the Haskell notation “(a, b)”. More generally, each category can have its own product construction, and the category’s “objects”/types do not need to have kind $*$ (classifying values). Both forms of added generality are quite useful, but they add complexity that would distract from the main topic of this paper (though see Section 12). The *Cartesian* operations must satisfy a universal property:

$$\forall h.\ \ h \equiv f \vartriangle g \iff \mathit{exl} \circ h \equiv f \wedge \mathit{exr} \circ h \equiv g$$

The names of these operators and style of writing the universal property, as well as those below, are adopted from (Gibbons 2002). A “terminal” category k has a designated object $\mathbf{1}$ (corresponding to Haskell’s () type), such that there is exactly one k-arrow from a to $\mathbf{1}$ for any object a in k:

```
class Category k ⇒ Terminal k where
  it :: a ‘k‘ 1
```

```
instance Terminal (→) where
  it = λa → ()
```

Finally, a *cartesian closed* category (CCC) adds “exponential” types $a \Mapsto b$ (representing morphisms as values/objects) with three operations:

```
class Cartesian k ⇒ Closed k where
  apply   :: ((a ⤇ b) × a) ‘k‘ b
  curry   :: ((a × b) ‘k‘ c) → (a ‘k‘ (b ⤇ c))
  uncurry :: (a ‘k‘ (b ⤇ c)) → ((a × b) ‘k‘ c)
```

```
instance Closed (→) where
  apply (f, x) = f x
  curry f      = λa b → f (a, b)
  uncurry g    = λ(a, b) → f a b
```

The notation “(⤇)” is a synonym for (→) in this paper but serves to remind us that each CCC can have its own notion of exponentials. The universal property:

$$\mathit{apply} \circ (\mathit{curry}\ f \circ \mathit{exl} \vartriangle \mathit{exr}) \equiv f$$

We will also want to consider categorical re-interpretations of some primitive constants. For base-typed primitives such as booleans and numbers, we’ll use arrows from terminal objects:

```
class Terminal k ⇒ ConstCat k b where
  unitArrow :: b → (1 ‘k‘ b)
```

```
instance ConstCat (→) b where
  unitArrow b = λ() → b
```

It’s sometimes more convenient to allow non-terminal domains:

```
const :: ConstCat k b ⇒ b → (a ‘k‘ b)
const b = unitArrow b ∘ it
```

Function-valued primitives may have interpretations in other categories, which can be captured in additional ad hoc *Category* subclasses. For instance,

```
class Cartesian k ⇒ BoolCat k where
  notC :: Bool ‘k‘ Bool
  andC, orC :: (Bool × Bool) ‘k‘ Bool
class NumCat k a where
  negateC :: a ‘k‘ a
  addC, mulC :: (a × a) ‘k‘ a
  ...
```

```
instance BoolCat (→) where
  notC = ¬
  andC = uncurry (∧); orC = uncurry (∨)
instance Num a ⇒ NumCat (→) a where
  negateC = negate
  addC = uncurry (+); mulC = uncurry (∗)
  ...
```

In a more general setting, different categories can have different *Bool*s. Note that primitives are in uncurried form.

3 CHANGING VOCABULARIES: CARTESIAN CLOSED CATEGORIES

The first step in compiling to categories is a syntactic transformation that converts the language of the simply typed λ-calculus (STLC) to a particular point-free form, corresponding to the vocabulary of cartesian closed categories (CCCs). For expository purposes, we will stay within the function category. The generalization to other categories will come in Section 4.

In the translation as described below, there are no typing contexts, and instead, every translated term is an explicit λ-abstraction. (See (Chu-Carroll 2012) for a description with typing contexts.) This style is especially convenient for use from within GHC's simplifier (Section 5), since the simplifier does not provide typing contexts. Translation occurs via a small collection of equivalences, presented below, to be used as rewrite rules, with each one taking us closer to a pure CCC expression. The λ-calculus terms are expressed in Haskell notation. Since we are translating function-typed terms, we can assume that we have an explicit abstraction, $\lambda(x :: \tau) \rightarrow U$ for some term U; otherwise, simply η-expand. Thus we need consider only a small number of cases—one for each kind of lambda term that appears in the body of a λ-abstraction.

First consider the case that the abstraction body is a variable. Since our terms are closed and well-typed, there is only one possible choice of variable, so we must have the identity function on τ:

$$(\lambda x \rightarrow x) \equiv id :: \tau \rightarrow \tau$$

Translating an application (as abstraction body) is a little more involved, involving the *Category*, *Cartesian*, and *Closed* instances for functions:

$$\begin{array}{ll}
 & \lambda x \rightarrow U\ V \\
\equiv & \{\text{- definition of } apply \text{ on } (\rightarrow) \text{ -}\} \\
 & \lambda x \rightarrow apply\ (U, V) \\
\equiv & \{\text{- definition of } (\triangle) \text{ on } (\rightarrow) \text{ -}\} \\
 & \lambda x \rightarrow apply\ (((\lambda x \rightarrow U) \triangle (\lambda x \rightarrow V))\ x) \\
\equiv & \{\text{- definition of } (\circ) \text{ on } (\rightarrow) \text{ -}\} \\
 & apply \circ ((\lambda x \rightarrow U) \triangle (\lambda x \rightarrow V))
\end{array}$$

If the body of an outer abstraction is another abstraction, we can curry a translation of the uncurried form:

$$\begin{array}{ll}
 & \lambda x \rightarrow \lambda y \rightarrow U \\
\equiv & \{\text{- definition of } curry \text{ on } (\rightarrow) \text{ -}\} \\
 & curry\ (\lambda(x, y) \rightarrow U)
\end{array}$$

For **case** expressions, suppose the scrutinee expression has a product type:

$$(\lambda x \rightarrow \mathbf{case}\ scrut\ \mathbf{of}\ \{(u, v) \rightarrow rhs\}) \equiv (\lambda x \rightarrow \mathbf{let}\ \{w = scrut; u = exl\ z; v = exr\ z\}\ \mathbf{in}\ rhs)$$

These **let** bindings are often then eliminated by GHC's simplifier. Other single-constructor data types are handled by conversion (Section 9). Translating multi-constructor data types requires a distributive category (Section 8).

The remaining case is a constant as abstraction body, i.e., $\lambda x \rightarrow c$. There are two possibilities. For simple types like *Bool* and *Int*, use *const*:

$$(\lambda x \rightarrow c) \equiv const\ c$$

If c has function type and a categorical interpretation via *BoolCat*, *NumCat*, etc, translate it accordingly:

$$\begin{array}{ll}
(\lambda x \rightarrow \neg) & \equiv constFun\ notC \\
(\lambda x \rightarrow (\wedge)) & \equiv constFun\ (curry\ andC) \\
\ldots &
\end{array}$$

where

$$\begin{array}{l} \mathit{constFun} :: \mathit{Closed}\ k \Rightarrow (a\ `k`\ b) \to (z\ `k`\ (a \Rrightarrow b)) \\ \mathit{constFun}\ f = \mathit{curry}\ (f \circ \mathit{exr}) \end{array}$$

Types involved in defining *constFun*:

$$\begin{array}{rl} f & :: a\ `k`\ b \\ f \circ \mathit{exr} & :: (z \times a)\ `k`\ b \\ \mathit{curry}\ (f \circ \mathit{exr}) & :: z\ `k`\ (a \Rrightarrow b) \end{array}$$

To see the translation to CCC form in practice, consider the following definitions:

$$\begin{array}{l} \mathit{sqr} :: \mathit{Num}\ a \Rightarrow a \to a \\ \mathit{sqr}\ a = a * a \\[4pt] \mathit{magSqr} :: \mathit{Num}\ a \Rightarrow a \times a \to a \\ \mathit{magSqr}\ (a, b) = \mathit{sqr}\ a + \mathit{sqr}\ b \\[4pt] \mathit{cosSinProd} :: \mathit{Floating}\ a \Rightarrow a \times a \to a \times a \\ \mathit{cosSinProd}\ (x, y) = (\cos z, \sin z)\ \textbf{where}\ z = x * y \end{array}$$

Type-specialized to a primitive type like *Double*, and converted to CCC form:

$$\begin{array}{l} \mathit{sqr} = \mathit{mulC} \circ (\mathit{id} \vartriangle \mathit{id}) \\ \mathit{magSqr} = \mathit{addC} \circ (\mathit{mulC} \circ (\mathit{exl} \vartriangle \mathit{exl}) \vartriangle \mathit{mulC} \circ (\mathit{exr} \vartriangle \mathit{exr})) \\ \mathit{cosSinProd} = (\mathit{cosC} \vartriangle \mathit{sinC}) \circ \mathit{mulC} \end{array}$$

The *sqr* conversion, step-by-step:

$$\begin{array}{ll} & \mathit{sqr}\ @\mathit{Double} \\ \equiv & \{\text{- inlining and simplifications by GHC -}\} \\ & \lambda x \to \mathit{timesDouble}\ x\ x \\ \equiv & \{\text{- translate application -}\} \\ & \mathit{apply} \circ ((\lambda x \to \mathit{timesDouble}\ x) \vartriangle (\lambda x \to x)) \\ \equiv & \{\text{- } \eta \text{ reduction by GHC -}\} \\ & \mathit{apply} \circ (\mathit{timesDouble} \vartriangle (\lambda x \to x)) \\ \equiv & \{\text{- translate } \mathit{timesDouble} \text{ and variable -}\} \\ & \mathit{apply} \circ (\mathit{curry}\ \mathit{mulC} \vartriangle \mathit{id}) \\ \equiv & \{\text{- Law for closed categories -}\} \\ & \mathit{mulC} \circ (\mathit{id} \vartriangle \mathit{id}) \end{array}$$

That last law, $\mathit{apply} \circ (\mathit{curry}\ h \vartriangle g) \equiv h \circ (\mathit{id} \vartriangle g)$, is a consequence of the universal property for exponentials. This law, like many others, is implemented as a GHC rewrite rules in syntactic form (Peyton Jones et al. 2001). Without such optimizations, *cosSinProd* becomes $(\mathit{cosC} \vartriangle \mathit{sinC}) \circ \mathit{mulC} \circ (\mathit{exl} \vartriangle \mathit{exr})$.

4 CHANGING CATEGORIES: CLOSED CARTESIAN FUNCTORS

In the previous section, we saw how to re-express STLC programs (with pairing and constants) into categorical vocabulary without changing their meaning. The value in doing so is that it then becomes easy to generalize beyond the original $(\to)$ category to other categories. Each such other category becomes an alternative interpretation of the STLC, and hence of Haskell programs, as we will see. To switch from $(\to)$, simply replace the $(\to)$ instances of all categorical operations in the translation above with corresponding instances for another category k.

The consistent replacement of interpretations while keeping vocabulary intact is equivalent to applying a *homomorphism*. Figure 1 shows the homomorphism properties for *Category*, *Cartesian*, and *Closed*.[1] Note that

[1]The properties are also known as "functor", "cartesian functor", and "closed cartesian functor", respectively. Note that Haskell's standard library comes with a *Functor* type class, but it is restricted to endofunctors on the standard Haskell category (i.e., from $(\to)$ to $(\to)$).

$$
\begin{aligned}
\mathcal{H}\ \mathit{id} &\equiv \mathit{id} \\
\mathcal{H}\ (g \circ f) &\equiv \mathcal{H}\ g \circ \mathcal{H}\ f
\end{aligned}
$$

(a) *Category*

$$
\begin{aligned}
\mathcal{H}\ \mathit{exl} &\equiv \mathit{exl} \\
\mathcal{H}\ \mathit{exr} &\equiv \mathit{exr} \\
\mathcal{H}\ (f \vartriangle g) &\equiv \mathcal{H}\ f \vartriangle \mathcal{H}\ g
\end{aligned}
$$

(b) *Cartesian*

$$
\begin{aligned}
\mathcal{H}\ \mathit{apply} &\equiv \mathit{apply} \\
\mathcal{H}\ (\mathit{curry}\ f) &\equiv \mathit{curry}\ (\mathcal{H}\ f) \\
\mathcal{H}\ (\mathit{uncurry}\ g) &\equiv \mathit{uncurry}\ (\mathcal{H}\ g)
\end{aligned}
$$

(c) *Closed*

Fig. 1. Homomorphism properties

the identity and composition on the left are for one category, and on the right for another.

Together with the translation from STLC to CCC form, these homomorphism equations are the key to compiling to alternative categories, thus giving sound, non-standard interpretations of functional programs. Interpreted as rewrite rules (oriented left-to-right), the homomorphism equations spell out a simple, systematic transformation from one category (initially $(\rightarrow)$) to another. It is important to note that both of these translation steps are *syntactic* (source code) transformations. Although homomorphisms are *semantic* properties, their mechanical application requires access to and manipulation of syntax. For this reason, the general technique described in this paper is most naturally implemented as part of compilation rather than a shallow or deep DSL embedding.

5 TRANSFORMING GHC CORE TO CCC

The Glasgow/Glorious Haskell Compiler (GHC) compiles Haskell for execution on CPUs. One of the major design choices in implementing GHC is to translate the large source language (Haskell) to a much smaller language, called "GHC Core". The Core language is a typed λ-calculus with a powerful type system, namely "System FC" (System F with constraints) (Sulzmann et al. 2007).

Since Core has a far more expressive type system than the STLC, it is not immediately obvious that the translation from STLC to CCC form as described above is applicable. A few techniques are required for bridging this gap, the most important being monomorphization. Polymorphism is extremely useful in writing modular programs, but if the ultimate function being compiled is monomorphic, then the intermediate polymorphism can be removed by monomorphizing, i.e., inlining polymorphic definitions and substituting monotypes for the type variables involved. Due to the presence of polymorphic recursion in Haskell, monomorphizing does not always have a finite result, and so sometimes the translation fails to terminate. In the many cases in which monomorphization does succeed, it has the additional benefit of leading to very efficient code. For instance, all dictionaries (used to implement type classes) are statically eliminated by partial evaluation, as in (Jones 1994).

In practice, the transformation to CCC form and conversion from $(\rightarrow)$ to other CCCs are implemented as GHC rewrite rules (Peyton Jones et al. 2001). Transformation is guided by the presence of a pseudo-function:[2]

$$\mathit{ccc} :: (a \rightarrow b) \rightarrow (a\ `k`\ b)$$

One might expect there to be a constraint on k, such as *Closed*. Instead, the rewrite rules introduce their own constraints as needed, allowing flexibility and extensibility. For instance, some target categories are cartesian but not *cartesian closed*, so alternative translations are needed. Homomorphism rules similar to the $\mathcal{H}$ examples of Section 2 push applications of *ccc* inward, eventually disappearing, as in the rules for *id*, *exl*, *exr*, and *apply* above as well as the similar rules for primitives like *addC* and *mulC*. Other rules prepare for homomorphism applicability.

Operating inside of GHC's simplifier allows translation to be done one fragment at a time with many other useful simplifications being applied to the results. This synergy between *ccc*-specific transformation and more general transformations helped considerably in making the implementation simple, efficient, and effective.

Rather than dozens of small rewrite rules, most of the rewriting is done in the form of a single GHC "built-in rule", which is a Haskell-defined function of type *CoreExpr* → *Maybe CoreExpr*, injected into GHC's simplifier

[2]More generally, *ccc* would have the signature of a categorical functor, i.e., $(a \rightarrow b) \rightarrow (f\ a\ `k`\ f\ b)$. This additional flexibility complicates translation, but is quite useful (as alluded to in Section 12) and will be addressed in a later paper.

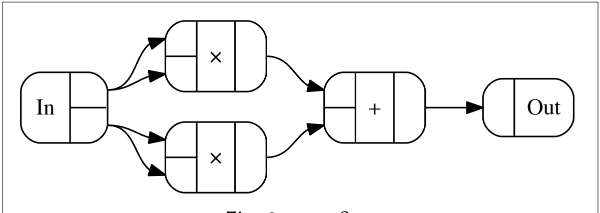

Fig. 2. *magSqr*

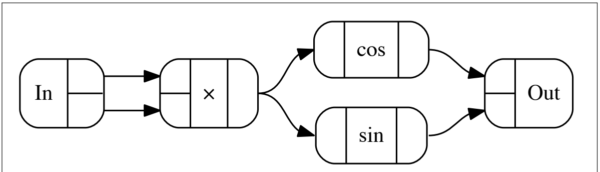

Fig. 3. *cosSinProd*

(Peyton Jones and Marlow 2002), whose job is to massage Core expressions into more efficient form. This choice loses some modularity while gaining efficiency via faster matching. More importantly, built-in rules remove some problematic limitations of the rewrite rule source language, including lack of side conditions.

6 CONSTRAINED CATEGORIES

The classes in Section 2 (*Category*, *Cartesian*, etc) are too simplistic for many useful target categories. It's often necessary to allow categories to restrict the domain and codomain types. For instance, general differentiable functions require a vector space structure, since derivative values are linear maps between vector spaces having a shared scalar field (Elliott 2009; Spivak 1971). Similarly, hardware generation requires representability of types as collections of wires. A similar issue was explored in (Sculthorpe et al. 2013a) for constrained monads, leading to a normal form for applicative and monadic computations that allowed use of the usual, unconstrained *Monad* class. It wasn't apparent, however, how to apply that solution to categories.

Since different categories will restrict their types ("objects") differently, let's add an associated type predicate (function from type to constraint) to the *Category* class from Section 2 (via the "constraint kinds" language extension (Bolingbroke 2011)) and use it to constrain the types involved:

```
infixr 9 ∘
class Category k where
  type Ok k a :: Constraint
  type Ok k a = ()   -- default vacuous constraint
  id  :: Ok k a ⇒ a ‘k‘ a
  (∘) :: Ok₃ k a b c ⇒ (b ‘k‘ c) → (a ‘k‘ b) → (a ‘k‘ c)
```

We'll see a lot of these *Ok* constraints, so define some helpers, as used in the signature for (∘):

```
type Ok₂ k a b   = (Ok k a, Ok k b)
type Ok₃ k a b c = (Ok₂ k a b, Ok k c)
 ...
```

The types involved in *Cartesian* and *Closed* methods are similarly constrained by *Ok*.

7 SOME APPLICATIONS

7.1 Computation graphs and compiling Haskell to hardware

It can be illuminating to visualize programs as computation graphs, revealing potential for parallel evaluation. For instance, the *magSqr* and *cosSinProd* examples in Section 3 can be visualized as in Figures 2 and 3.

Underlying these diagrams is a Kleisli-like category of directed graphs based on a simple state monad that supplies output ports and a list of instantiated primitive components.[3]

```
data Graph a b = Graph (Ports a → GraphM (Ports b))
type GraphM = State (Port, [Comp])
```

[3]We could instead combine *State Port* with *Writer* [*Comp*], but the form shown more easily extends optimizations discussed below.

```
data Comp = ∀a b.Comp String (Ports a) (Ports b)    -- op name, inputs, outputs
```

The port collections are represented as a generalized algebraic data type (GADT):

```
type Port   = Int

data Ports :: ∗ → ∗ where
  UnitP   :: Ports 1
  BoolP   :: Port → Ports Bool
  IntP    :: Port → Ports Int
  DoubleP :: Port → Ports Double
  PairP   :: Ports a → Ports b → Ports (a × b)
  FunP    :: Graph a b → Ports (a ⤇ b)
```

The *Category*, *Cartesian*, *Terminal*, and *Closed* instances are very like that of *Kleisli* (Hughes 1998), but they must also manage the isomorphisms between pair ports and port pairs and between function ports and *Graphs*:[4]

```
instance Category Graph where
  type Ok Graph a = GenPorts a
  id = Graph return
  Graph g ∘ Graph f = Graph (g <=< f)

instance Cartesian Graph where
  exl = Graph (λ(PairP a _) → return a)
  exr = Graph (λ(PairP _ b) → return b)
  Graph f △ Graph g = Graph (liftA2 (liftA2 PairP) f g)

instance Terminal Graph where
  it = Graph (const (return UnitP))

instance Closed Graph where
  apply = Graph (λ(PairP (FunP (Graph ab)) a) → ab a)
  curry (Graph f) = Graph (λa → return (FunP (Graph (λb → f (PairP a b)))))
  uncurry (Graph g) = Graph (λ(PairP a b) → do {FunP (Graph f) ← g a; f b})
```

All that remains is to define instances for primitive operations, each of which simply adds a component (graph node) defined by an operation name and a typed collections of ports for the inputs and the outputs.

These *Ports* structures are generated from their types, with no ports for **1**, one for *Bool*, *Int*, and *Double*, and pairs of recursively generated structures for pairs:

```
genPort :: GraphM Port    -- single port
genPort = do {(o, comps) ← get; put (o + 1, comps); return o}

class GenPorts a where genPorts :: GraphM (Ports a)

instance GenPorts 1      where genPorts = return UnitP
instance GenPorts Bool   where genPorts = fmap BoolP genPort
instance GenPorts Int    where genPorts = fmap IntP genPort
instance GenPorts Double where genPorts = fmap DoubleP genPort

instance (GenPorts a, GenPorts b) ⇒ GenPorts (a × b) where genPorts = liftA2 PairP genPorts genPorts
```

To add a new graph component, generate output ports by type, associate with the primitive and inputs, and to the accumulating component list:

[4]The "<=<" operator is Kleisli composition, of type $Monad\ m \Rightarrow (b \rightarrow m\ c) \rightarrow (a \rightarrow m\ b) \rightarrow (a \rightarrow m\ c)$.

```
genComp :: GenPorts b ⇒ String → Graph a b
genComp name = Graph (λpa → do {pb ← genPorts; modify (second (Comp name pa pb :)); return pb})
```

Now we have everything we need to easily instantiate the operation-specific classes:

```
instance BoolCat Graph where
  notC = genComp "¬"
  andC = genComp "∧"
  orC  = genComp "∨"
instance (Num a, GenPorts a) ⇒ NumCat Graph a where
  negateC = genComp "negate"
  addC    = genComp "+"
  subC    = genComp "-"
  mulC    = genComp "×"
 ...
```

The *Eq* and *Num* constraints aren't strictly necessary in this simple implementation, but they serve to remind us of the expected translation from *Eq* and *Num* methods.

Notice that the *Category*, *Cartesian*, and *Closed* methods produce no components. Instead, *Category* manages connectivity, *Cartesian* discards and replicates signals, and *Closed* generates sub-graphs.

The simple representation and implementation outlined above can be improved by adding a few optimizations. The simplest is constant folding: when an operation is fed by only constant components (having no inputs), perform the operation during graph construction, and generate another constant component. Going further, many other algebraic simplifications can be applied, such as addition with zero, multiplication by one, double negation, etc. Finally, redundant computation can be eliminated via hash-consing during graph construction, at the cost of tracking more information about the graph as it is being generated. In practice, these optimizations prove quite worthwhile (Elliott 2017).

Once a computation graph is constructed, it can be rendered into a picture, as in the illustrations in this paper. Additionally, graphs can be rendered into machine-readable circuit descriptions in a hardware description language such as Verilog or VHDL. There is a library operation (unknown to the compiler plugin) that generates a picture and a Verilog source file, roughly:

```
mkCircuit :: Ok₂ Graph a b ⇒ String → Graph a b → IO ()
```

To make a circuit, one applies the pseudo-function *ccc* to a monomorphic Haskell function and gives the result to *mkCircuit*, e.g., the *magSqr* example defined in Section 3, type-specialized to 32-bit integers:

```
main = mkCircuit "magSqr" (ccc (magSqr @Double))
```

Type inference determines the target category to be *Graph*, and the plugin's added transformation rules push the *ccc* application progressively inward, inlining where needed, with many of GHC's standard simplifications tidying things up along the way. When this *main* is compiled and run, it generates the picture in Figure 2 and the following Verilog implementation:

```
module magSqr (In_0, In_1, Out);
  input [31:0] In_0;
  input [31:0] In_1;
  output [31:0] Out;
  wire [31:0] Plus_I0;
  wire [31:0] Times_I3;
  wire [31:0] Times_I4;
  assign Plus_I0 = Times_I3 + Times_I4;
  assign Out = Plus_I0;
```

```
    assign Times_I3 = In_0 * In_0;
    assign Times_I4 = In_1 * In_1;
  endmodule
```

7.2 Syntax

Section 3 showed CCC expressions for three simple functions (*sqr*, *magSqr*, and *cosSinProd*). Those expressions were generated by interpreting the corresponding functions in a syntactic category, which we'll now see. Since CCC expressions can get large, we'll want multi-line pretty-printing, for which we can use a common library (Hughes 1995; Hughes and Peyton Jones 2007).

Start with a simple type of untyped syntax trees:

```
type DocTree = Tree PDoc
```

where *Tree a* is a standard type of rose trees having a value of type *a* at each node, and *PDoc* is a pretty-printing document, parametrized by contextual binding precedence (for inserting parentheses as needed):

```
data Tree a = Node a [Tree a]
type PDoc = Rational → Doc
```

One could use *String* in place of *PDoc* in the definition of *DocTree*, but *PDoc* allows for complex constant values that can be pretty-printed and parenthesized in context. CCC expressions are very simple, and so is pretty-printing, which handles infix operators and general applications:

```
prettyTree :: DocTree → PDoc
prettyTree (Node d [u, v]) p | Just (q, (lf, rf)) ← lookup name fixities =
  maybeParens (p > q) $ sep [prettyTree u (lf q) <+> text name, (prettyTree v) (rf q)]
  where name   = show (d 0)
        fixities = fromList [("∘", (9, assocRight)), ("△", (3, assocRight)), ("▽", (2, assocRight))]
prettyTree (Node f es) p = maybeParens (¬ (null es) ∧ p > 10) (sep (f 10 : map (λe → prettyTree e 11) es))
```

Operator fixity is represented by a number (with higher numbers for tighter binding and 10 for function application) together with a pair of functions that adjust for left and right arguments:

```
type Prec   = Rational
type Fixity = (Prec, Assoc)
type Assoc  = (Prec → Prec, Prec → Prec)

assocLeft, assocRight, assocNone :: Assoc
assocLeft  = (id, succ)
assocRight = (succ, id)
assocNone  = (succ, succ)
```

Next, wrap up these *untyped* expressions in a phantom-typed representation (Hinze 2003; Leijen and Meijer 1999), representing an arrow from *a* to *b*, and define some utility functions:

```
newtype Syn a b = Syn DocTree

atom :: Pretty a ⇒ a → Syn a b
atom a = Syn (Node (ppretty a) [])

appt :: String → [DocTree] → DocTree
appt s ts = Node (const (text s)) ts
```

```
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])
```

With these utilities in hand, categorical operations come easily, e.g.,

```
instance Category Syn where
  id  = app_0 "id"
  (∘) = app_2 "∘"

instance Cartesian Syn where
  exl = app_0 "exl"
  exr = app_0 "exr"
  (△) = app_2 "△"
```

```
instance Closed Syn where
  apply   = app_0 "apply"
  curry   = app_1 "curry"
  uncurry = app_1 "uncurry"
instance BoolCat Syn where
  andC = app_0 "andC"
  orC = app_0 "orC"
  ...
```

7.3 Products of categories

Why give only one interpretation to a functional program when we can give two, such as a graph *and* the corresponding syntactic form? One way is to compile twice, each with a different target category. A more convenient and efficient alternative is to compile once to a *product* of categories. The arrows of such a product is represented by an arrow from each category, acting completely independently:

```
infixl 7 ⊗
data (p ⊗ q) a b = p a b ⊗ q a b
```

Categorical operations simply combine the corresponding operations of each category, e.g.,

```
instance (Category k, Category k′) ⇒ Category (k ⊗ k′) where
  type Ok (k ⊗ k′) a = (Ok k a, Ok k′ a)
  id = id ⊗ id
  (g ⊗ g′) ∘ (f ⊗ f′) = (g ∘ f) ⊗ (g′ ∘ f′)
instance (Cartesian k, Cartesian k′) ⇒ Cartesian (k ⊗ k′) where
  exl = exl ⊗ exl
  exr = exr ⊗ exr
  (f ⊗ f′) △ (g ⊗ g′) = (f △ g) ⊗ (f′ △ g′)
 ...
```

As an identity for (⊗), we also have a category with exactly one arrow for each domain/codomain pair:

```
data U_2 a b = U_2
instance Category U_2 where
  id = U_2
  U_2 ∘ U_2 = U_2
instance Cartesian U_2 where
  exl = U_2
  exr = U_2
  U_2 △ U_2 = U_2
 ...
```

7.4 Linear maps

Although we usually represent functions as code, sometimes we can use data instead. For *linear* functions, one can instead use a very compact data representation. For instance, any linear function from $\mathbb{R}^2$ to $\mathbb{R}^3$ can be represented as a matrix of six numbers. Moreover, since the identity function is linear, and the composition of linear functions is linear, we have a category. For linearity to be meaningful, we need vector space over a scalar field (or just a (semi)module over a (semi)ring). There are various ways to formulate vector spaces over a scalar field s. A particularly elegant choice is that of *free vector spaces*, each of which is isomorphic to the space of functions $f :: A \rightarrow s$ from some index set A. Rather than mapping to functions, however, we can use a memoized form tries for these functions as compositions of some basic functor building blocks (Hinze 2000a).

Conversion to and from (representable endo)functor form is managed by the following class:

```
class HasV s a where
  type V s a :: ∗ → ∗   -- “vector form”
  toV  :: a → V s a s
  unV :: V s a s → a
```

Some instances:

```
instance HasV s 1 where
  type V s 1 = U₁
  toV ()   = U₁
  unV U₁ = ()
instance HasV Double Double where
  type V Double Double = Par₁
  toV  = Par₁
  unV = unPar1
instance (HasV s a, HasV s b) ⇒ HasV s (a × b) where
  type V s (a × b) = V s a × V s b
  toV (a, b)     = toV a × toV b
  unV (f × g) = (unV f, unV g)
```

The Par_1, U_1 and (×) type constructors are identity functor, unit functor, and cartesian functor product, taken from *GHC.Generics* (Magalhães et al. 2010; Magalhães et al. 2011):

```
data U₁        s = U₁
data Par₁      s = Par₁ s
data (f × g) s = f g × g s
```

A linear map from a to b is represented by a vector of vectors, i.e., a “matrix”, in row-major form (though we could as easily use column-major):

```
newtype a ⊸ₛ b = LMap (V s b (V s a s))
```

The categorical instances are as follows, with auxiliary definitions given in Appendix A:

```
instance Num s ⇒ Category (⊸ₛ) where
  type Ok (⊸ₛ) a = (HasV s a, OkLF (V s a))
  id = LMap idL
  LMap g ∘ LMap f = LMap (g ‘compL‘ f)
instance Num s ⇒ Cartesian (⊸ₛ) where
  exl = LMap exlL
  exr = LMap exrL
  LMap g △ LMap f = LMap (g ‘forkL‘ f)
instance Num s ⇒ Terminal (⊸ₛ) where
  it = LMap U₁
```

Linear map *application* is a functor from $(\multimap_s)$ to $(\rightarrow)$, serving as a semantic function for $(\multimap_s)$:

$$\begin{array}{l} \mathit{lapply} :: (\mathit{Ok}_2\ (\multimap_s)\ a\ b, \mathit{Num}\ s) \Rightarrow (a \multimap_s b) \rightarrow (a \rightarrow b) \\ \mathit{lapply}\ (\mathit{LMap}\ \mathit{ba}) = \mathit{unV} \circ \mathit{lapplyL}\ \mathit{ba} \circ \mathit{toV} \end{array}$$

Conversely, given a function $f :: a \rightarrow b$, we can construct a linear map $f' :: a \multimap_s b$ such that $\mathit{lapply}\ f' \equiv f$ if f is linear:

$$\begin{array}{l} \mathit{linear} :: (\mathit{Ok}_2\ (\multimap_s)\ a\ b, \mathit{HasL}\ (V\ s\ a), \mathit{Num}\ s) \Rightarrow (a \rightarrow b) \rightarrow (a \multimap_s b) \\ \mathit{linear}\ h = \mathit{LMap}\ (\mathit{linearL}\ (\mathit{toV} \circ h \circ \mathit{unV})) \end{array}$$

7.5 Automatic differentiation

Next, let's consider how to differentiate functions exactly. To handle multi-dimensional types, assume that our functions map between free vector spaces over a common scalar field. In this general setting, derivative values are linear maps (Elliott 2009; Spivak 1971). We thus have the following type for differentiation:

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

Although *deriv f* is not computable from the function *f*, we can construct it homomorphically from a categorical recipe for *f* by compiling to a suitable category. Consider the chain rule in terms of derivatives as linear maps (Spivak 1971, Theorem 2-2):

$$\mathit{deriv}\ (g \circ f)\ a \equiv \mathit{deriv}\ g\ (f\ a) \circ \mathit{deriv}\ f\ a$$

While the composition in the left side of the chain rule is on functions, the composition on the right is on linear maps. The notion of derivatives as linear maps subsumes various representations including scalar values, vectors, covectors, matrices, and higher dimensional counterparts. Likewise, this one general chain rule subsumes many specific variations involving scalar multiplication, inner products, outer products, matrix products, Hessians, etc.

Note that the derivative of $g \circ f$ depends not only on the derivatives of *g* and *f*, but also on *g* itself, so *deriv* is *not* a functor. All is not lost though, as we can instead compositionally construct a combination of functions *and* their derivatives. A straightforward pairing of the two leads to the following representation of differentiable functions between vector spaces over a field *s*:

$$\textbf{data}\ a \leadsto_s b = D\ (a \rightarrow b)\ (a \rightarrow (a \multimap_s b))) \quad \text{-- first try}$$

This representation, however, prevents exploiting the considerable amount of computation that functions and their derivatives typically have in common. Fortunately, there is a simple solution, using the $(\vartriangle)$ isomorphism from *Cartesian* to combine functions and their derivatives:

$$\textbf{data}\ a \leadsto_s b = D\ \{\ \mathit{unD} :: a \rightarrow b \times (a \multimap_s b)\ \} \quad \text{-- allows work sharing}$$

Our goal is to implement the following functor

$$\begin{array}{l} \mathit{andDeriv} :: (a \rightarrow b) \rightarrow (a \leadsto_s b) \\ \mathit{andDeriv}\ f = D\ (f \vartriangle \mathit{deriv}\ f) \quad \text{-- specification} \end{array}$$

This combination forms a local affine (first order) approximation of *f* at every point in its domain. Once we implement *andDeriv* faithfully to this specification, we will have a simple implementation of *deriv*:

$$\mathit{deriv}\ f = \mathit{snd} \circ \mathit{unD}\ (\mathit{andDeriv}\ f)$$

Linear functions are especially easy to differentiate, since they are their own (perfect) linear approximations. The following helper function takes a linear function and its linear map counterpart:[5]

$$\begin{array}{l} \mathit{linearD} :: (\mathit{Num}\ s, \mathit{Ok}_2\ (\multimap_s)\ a\ b) \Rightarrow (a \rightarrow b) \rightarrow (a \multimap_s b) \rightarrow (a \leadsto_s b) \\ \mathit{linearD}\ f\ f' = D\ (f \vartriangle \mathit{const}\ f') \end{array}$$

[5]One could compute either the function or the map from the other, using *lapply* or *linear* above, but the two-argument *linearD* allows for a useful generalization below.

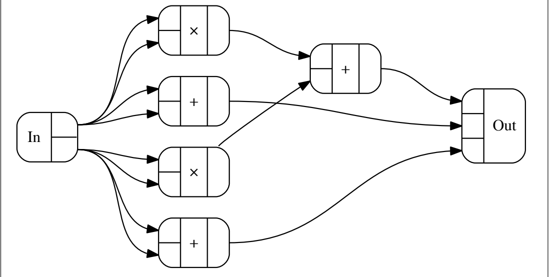

Fig. 4. *andDeriv magSqr*

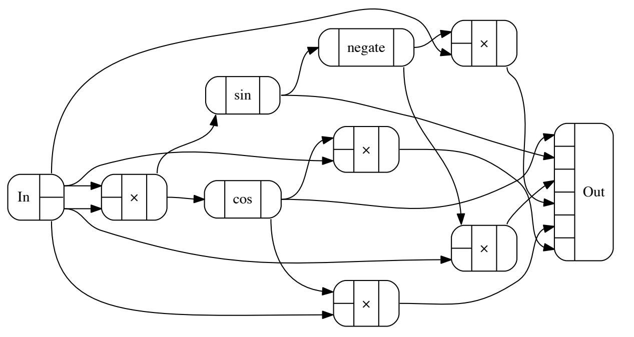

Fig. 5. *andDeriv cosSinProd*

Now we're ready to define the *Category* instance for differentiable functions. Since the identity function is linear, *linearD* gives us the differentiable version. Composition follows from the chain rule:

$$
\begin{array}{l}
\textbf{instance } \mathit{Num}\ s \Rightarrow \mathit{Category}\ (\leadsto_s)\ \textbf{where} \\
\quad \textbf{type } \mathit{Ok}\ (\leadsto_s)\ a = \mathit{Ok}\ (\multimap_s)\ a \\
\quad \mathit{id} = \mathit{linearD}\ \mathit{id}\ \mathit{id} \\
\quad D\ g \circ D\ f = D\ (\lambda a \to \textbf{let}\ \{(b, f') = f\ a; (c, g') = g\ b\}\ \textbf{in}\ (c, g' \circ f'))
\end{array}
$$

Product operations are handled similarly. Since *exl* and *exr* are both linear, we can again use *linearD*. For ($\triangle$), we'll need a counterpart to the chain rule:

$$\mathit{deriv}\ (f \triangle g)\ a = \mathit{deriv}\ f\ a \triangle \mathit{deriv}\ g\ a$$

Assembling the pieces,

$$
\begin{array}{l}
\textbf{instance } \mathit{Num}\ s \Rightarrow \mathit{Cartesian}\ (\leadsto_s)\ \textbf{where} \\
\quad \mathit{exl} = \mathit{linearD}\ \mathit{exl}\ \mathit{exl} \\
\quad \mathit{exr} = \mathit{linearD}\ \mathit{exr}\ \mathit{exr} \\
\quad D\ f \triangle D\ g = D\ (\lambda a \to \textbf{let}\ \{(b, f') = f\ a; (c, g') = g\ a\}\ \textbf{in}\ ((b, c), f' \triangle g'))
\end{array}
$$

Knowledge of derivatives for numerical operations lives in instances of *NumCat*, *FloatingCat*, etc:

$$
\begin{array}{l}
\textbf{instance } (\mathit{Num}\ s, V\ s\ s \sim \mathit{Par}_1, \mathit{Ok}\ (\multimap_s)\ s) \Rightarrow \mathit{NumCat}\ (\leadsto_s)\ s\ \textbf{where} \\
\quad \mathit{negateC} = \mathit{linearD}\ \mathit{negateC}\ (\mathit{linear}\ \mathit{negateC}) \\
\quad \mathit{addC} \quad\ = \mathit{linearD}\ \mathit{addC}\ (\mathit{linear}\ \mathit{addC}) \\
\quad \mathit{mulC} \quad\ = D\ (\mathit{mulC} \triangle \lambda(a, b) \to \mathit{linear}\ (\lambda(da, db) \to da * b + db * a)) \\
\quad \ldots
\end{array}
$$

Figures 4 and 5 shows the result of *andDeriv* on the *magSqr* and *cosSinProd* examples from Section 3. Because *magSqr* :: $\mathbb{R}^2 \to \mathbb{R}$, its derivative values have type $\mathbb{R}^2 \multimap_{\mathbb{R}} \mathbb{R}$, represented by a pair of reals. Because *cosSinProd* :: $\mathbb{R}^2 \to \mathbb{R}^2$, its derivative values have type $\mathbb{R}^2 \multimap_{\mathbb{R}} \mathbb{R}^2$, represented by a *pair of pairs* of reals. Figures 4 and 5 shows the results of *deriv* on the same two examples.

The *Category* and *Cartesian* instances for ($\leadsto_s$) rely on ($\multimap_s$) *only* for its instances of these same classes. Thanks to this simple relationship, we can easily generalize from ($\multimap_s$) to an arbitrary bicartesian category, re-specializing to ($\leadsto_s$):

$$
\begin{array}{l}
\textbf{newtype } \mathit{GD}\ k\ a\ b = D\ (a \to b \times (a\ `k`\ b)) \\
\textbf{type } (\leadsto_s) = \mathit{GD}\ (\multimap_s)
\end{array}
$$

The instances for *GD k* are as they were for *D* above, except for adding the properties required of *k*:

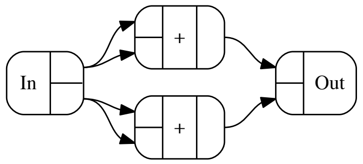

Fig. 6. *deriv magSqr*

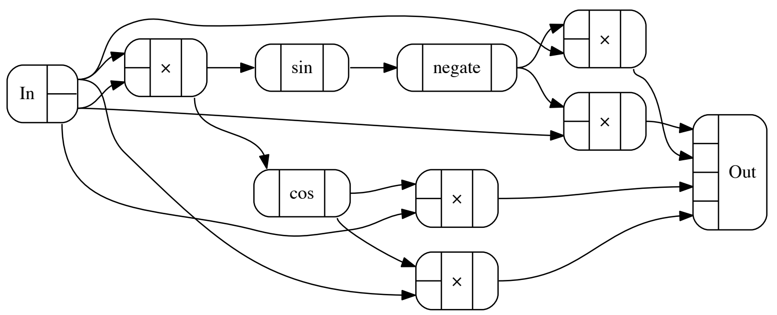

Fig. 7. *deriv cosSinProd*

instance *Category k* ⇒ *Category* (*GD k*) **where** ...
instance *Cartesian k* ⇒ *Cartesian* (*GD k*) **where** ...

7.6 Incremental computation

When a function is applied to the same argument twice, it performs the same work, unless the function is memoized. When a function is applied to two *similar* arguments, memoization is no help, and the second application must start from scratch even though some of the work may be repeated between the two applications. The idea of incremental computation (IC) is to make a small amount of extra effort on the first invocation so that invocations on similar arguments may be done *incrementally* and thus inexpensively.

To formulate IC, first define an interface for incremental changes to values, mimicking (Cai et al. 2014):

```
infixl 6 ⊖, ⊕
class HasDelta a where
  type Δ a
  (⊕) :: a → Δ a → a
  (⊖) :: a → a → Δ a
  0    :: Δ a
```

The unit type has a trivial instance, since there can be no changes. One possibility for atomic types is to represent changes as a possible new value:

```
instance HasDelta Int where
  type Δ Int = Maybe Int
  (⊕) = fromMaybe
  new ⊖ old = if new ≡ old then Nothing else Just new
  0 = Nothing
```

A change for pairs is a pair of changes:

```
instance (HasDelta a, HasDelta b) ⇒ HasDelta (a × b) where
  type Δ (a × b) = Δ a × Δ b
  (a, b) ⊕ (da, db) = (a ⊕ da, b ⊕ db)
  (a′, b′) ⊖ (a, b) = (a′ ⊖ a, b′ ⊖ b)
  0 = (0, 0)
```

Finally, a change for functions is a function *to* changes:

```
instance HasDelta b ⇒ HasDelta (a → b) where
  type Δ (a → b) = a → Δ b
```

$$
\begin{aligned}
&(\oplus) = \mathit{liftA}_2\ (\oplus)\\
&(\ominus) = \mathit{liftA}_2\ (\ominus)\\
&0 = \mathit{const}\ 0
\end{aligned}
$$

A change morphism says how to map changes to changes:

$\mathbf{newtype}\ \mathit{DelX}\ a\ b = \mathit{DelX}\ (\Delta\ a \to \Delta\ b)$

It's easy to then give *Category* and *Cartesian* instances for *DelX*:

$$
\begin{array}{l|l}
\mathbf{instance}\ \mathit{Category}\ \mathit{DelX}\ \mathbf{where} & \mathbf{instance}\ \mathit{Cartesian}\ \mathit{DelX}\ \mathbf{where}\\
\quad \mathbf{type}\ \mathit{Ok}\ \mathit{DelX} = \mathit{HasDelta} & \quad \mathit{exl} = \mathit{DelX}\ \mathit{exl}\\
\quad \mathit{id} = \mathit{DelX}\ \mathit{id} & \quad \mathit{exr} = \mathit{DelX}\ \mathit{exr}\\
\quad \mathit{DelX}\ g \circ \mathit{Del}\ f = \mathit{Del}\ (g \circ f) & \quad \mathit{DelX}\ f \vartriangle \mathit{Del}\ g = \mathit{Del}\ (f \vartriangle g)
\end{array}
$$

Since *DelX* is a cartesian functor, we can use it with generalized automatic differentiation to get a category of incremental computation:

$\mathbf{type}\ \mathit{Inc} = \mathit{GD}\ \mathit{DelX}$

An "atomic type" is one for which changes are all or nothing, as with *Int* above. Such types have simple and useful ways of constructing change arrows and incremental functions:

$$
\begin{aligned}
&\mathbf{type}\ \mathit{Atomic}\ a = (\mathit{HasDelta}\ a, \Delta\ a \sim \mathit{Maybe}\ a)\\
&\mathit{atomic}_1 :: (\mathit{Atomic}\ a, \mathit{Atomic}\ b) \Rightarrow (a \to b) \to \mathit{Inc}\ a\ b\\
&\mathit{atomic}_1\ f = D\ (\lambda a \to (f\ a, \mathit{DelX}\ (\lambda \mathbf{case}\ \mathit{Nothing} \to \mathit{Nothing}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad d \qquad\quad \to \mathit{Just}\ (f\ (a \oplus d)))))\\
&\mathit{atomic}_2 :: (\mathit{Atomic}\ a, \mathit{Atomic}\ b, \mathit{Atomic}\ c) \Rightarrow (a \times b \to c) \to \mathit{Inc}\ (a \times b)\ c\\
&\mathit{atomic}_2\ f = D\ (\lambda ab \to (f\ ab, \mathit{DelX}\ (\lambda \mathbf{case}\ (\mathit{Nothing}, \mathit{Nothing}) \to \mathit{Nothing}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad d \qquad\qquad\qquad\quad \to \mathit{Just}\ (f\ (ab \oplus d)))))
\end{aligned}
$$

Now it's easy to define numeric operations on atomic types, e.g.,

$$
\begin{aligned}
&\mathbf{instance}\ (\mathit{Atomic}\ s, \mathit{Num}\ s) \Rightarrow \mathit{NumCat}\ \mathit{Inc}\ s\ \mathbf{where}\\
&\quad \mathit{negateC} = \mathit{atomic}_1\ \mathit{negateC}\\
&\quad \mathit{addC} \quad\ = \mathit{atomic}_2\ \mathit{addC}\\
&\quad \mathit{subC} \quad\ = \mathit{atomic}_2\ \mathit{subC}\\
&\quad \mathit{mulC} \quad\ = \mathit{atomic}_2\ \mathit{mulC}
\end{aligned}
$$

7.7 Interval analysis

Interval analysis (IA) is a technique for computing bounds on functions, mapping domain intervals to codomain intervals (Moore 1966), with applications including root finding, minimization, and error management. Given a function f, a corresponding interval function $\hat{f}$ has the property that for any domain interval I, $\forall x \in I.\ f\ x \in \hat{f}\ I$. The compositional nature of IA makes it a natural fit for a categorical interface. Different types have different interval representations, with atomic values having lower and upper bound, while product intervals are products of intervals ("boxes"), and function intervals are functions between intervals:

$$
\begin{aligned}
&\mathbf{data}\ \mathit{IFun}\ a\ b = \mathit{IFun}\ (\mathit{Interval}\ a \to \mathit{Interval}\ b)\\
&\mathbf{type}\ \mathit{family}\ \mathit{Interval}\ a\\
&\mathbf{type\ instance}\ \mathit{Interval}\ \mathit{Double} \ \ = \mathit{Double} \times \mathit{Double}\\
&\mathbf{type\ instance}\ \mathit{Interval}\ \mathit{Int} \qquad = \mathit{Int} \times \mathit{Int}\\
&\mathbf{type\ instance}\ \mathit{Interval}\ (a \times b) = \mathit{Interval}\ a \times \mathit{Interval}\ b\\
&\mathbf{type\ instance}\ \mathit{Interval}\ (a \to b) = \mathit{Interval}\ a \to \mathit{Interval}\ b
\end{aligned}
$$

Instances for the basic category classes are as simple as can be:

```
instance Category IFun where
  id = IFun id
  IFun g ∘ IFun f = IFun (g ∘ f)
instance Cartesian IFun where
  exl = IFun exl
  exr = IFun exr
  IFun f △ IFun g = IFun (f △ g)
```

```
instance Closed IFun where
  apply = IFun apply
  curry (IFun f) = IFun (curry f)
  uncurry (IFun g) = IFun (uncurry g)

instance Interval b ~ (b × b) ⇒ ConstCat IFun b where
  unitArrow b = IFun (unitArrow (b, b))
```

The real work is done in numeric operations:

```
instance (Interval a ~ (a × a), Num a, Ord a) ⇒ NumCat IFun a where
  addC = IFun (λ((a_lo, a_hi), (b_lo, b_hi)) → (a_lo + b_lo, a_hi + b_hi))
  mulC = IFun (λ((a_lo, a_hi), (b_lo, b_hi)) → let cs = [a_lo ∗ b_lo, a_lo ∗ b_hi, a_hi ∗ b_lo, a_hi ∗ b_hi] in
                                                (minimum cs, maximum cs))
  ...
```

7.8 Other examples

Kmett (2011) shows how to form a simple cartesian category of entailments between Haskell's type constraints. The natural formulation of this category has kind *Constraint* → *Constraint* → ∗, relying on the poly-kinded generalization mentioned in Section 12. (It can be encoded somewhat less directly via values that can be converted to and from constraint dictionaries, similarly to the conversion to and from functor representations of linear maps in Section 7.4). This category is useful for boosting the power of GHC's type inference, particularly to overcome the lack of universally quantified constraints. Other deductive systems may be possible as well, including disjunction (coproduct) and implication (exponential).

Just as linear maps form a (bi)cartesian category, so do polynomials (including product domains and ranges). As long as one uses only addition and multiplication as primitives, functional programs can be compiled into polynomials, which can then be analyzed efficiently and exactly, finding roots, minima and maxima, derivatives, and integrals, as well as evaluated efficiently in parallel using parallel prefix algorithms (Blelloch 1990; Elliott 2017). Power series (infinite polynomials) can probably be treated the same way (allowing operations beyond addition and multiplication), perhaps using the elegantly defined operations from (McIlroy 1999).

8 COCARTESIAN AND DISTRIBUTIVE CATEGORIES

Although not used in the examples of this paper, another common and useful concept is that of *cocartesian* categories. The interface is exactly dual to that of *Cartesian*, with sums in place of cartesian products, having two introduction and one elimination operation:

```
infixr 2 ▽
class Category k ⇒ Cocartesian k where
  inl :: Ok2 k a b ⇒ a ‘k‘ (a + b)    -- inject left
  inr :: Ok2 k a b ⇒ b ‘k‘ (a + b)    -- inject right
  (▽) :: Ok3 k a c d
      ⇒ (c ‘k‘ a) → (d ‘k‘ a) → ((c + d) ‘k‘ a)    -- “join”
```

```
instance Cocartesian (→) where
  inl = Left
  inr = Right
  (f ▽ g) (Left  a) = f a
  (f ▽ g) (Right b) = g b
```

The universal property is also dual to that of *Cartesian*:

$$\forall h.\ \ h \equiv f \triangledown g \Longleftrightarrow h \circ inl \equiv f \wedge h \circ inr \equiv g$$

Again, details are adopted from (Gibbons 2002). As with products, it will be useful to generalize the notion of coproducts beyond sum types. A “bicartesian” category is both cartesian and cocartesian.

A *distributive* category is one that enables distribution of products over coproducts:

```
class (Cartesian k, Cocartesian k) ⇒ Distributive k where
  distl :: Ok₃ k a u v
        ⇒ (a × (u + v)) ‘k‘ (a × u + a × v)
  distr :: Ok₃ k a u v
        ⇒ (a × u + a × v) ‘k‘ (a × (u + v))
```

```
instance DistribCat (→) where
  distl (a, Left  u) = Left  (a, u)
  distl (a, Right v) = Right (a, v)
  distr (Left  u, b) = Left  (u, b)
  distr (Right v, b) = Right (v, b)
```

We can define either *distl* or *distr* in terms of the other (exercise), so only one need be primitive.

Distributive categories enable translation of definition by cases. Consider only **case** over binary sums $a + b$ for now. Other multi-constructor data types will be translated into binary sums, as described in Section 9. Transform such expressions (in context) as follows:

$$(\lambda x \rightarrow \textbf{case}\ \mathit{scrut}\ \textbf{of}\ \{\mathit{Left}\ u \rightarrow U; \mathit{Right}\ v \rightarrow V\}) \equiv (\lambda x \rightarrow (U \triangledown V)\ \mathit{scrut})$$

We already know how to handle applications under an outer abstraction. For the rest,

$$(\lambda x \rightarrow U \triangledown V) \equiv \mathit{curry}\ ((\mathit{uncurry}\ (\lambda x \rightarrow U) \triangledown \mathit{uncurry}\ (\lambda x \rightarrow V)) \circ \mathit{distl})$$

Proof: From the type of $(\triangledown)$, we must have $U :: u \rightarrow c$ and $V :: v \rightarrow c$ for some types u, v, and c, and $(\lambda x \rightarrow U \triangledown V) :: \tau \rightarrow u + v \rightarrow c$. Consider two cases, first applying both sides to x and *Left* u:

$$\begin{aligned}
&\ (\lambda x \rightarrow U \triangledown V)\ x\ (\mathit{Left}\ u) \\
\equiv&\ (U \triangledown V)\ (\mathit{Left}\ u) \\
\equiv&\ U\ u \\
\equiv&\ (\lambda x \rightarrow U)\ x\ u \\
\equiv&\ \mathit{uncurry}\ (\lambda x \rightarrow U)\ (x, u) \\
\equiv&\ (\mathit{uncurry}\ (\lambda x \rightarrow U) \triangledown \mathit{uncurry}\ (\lambda x \rightarrow V))\ (\mathit{Left}\ (x, u)) \\
\equiv&\ (\mathit{uncurry}\ (\lambda x \rightarrow U) \triangledown \mathit{uncurry}\ (\lambda x \rightarrow V))\ (\mathit{distl}\ (x, \mathit{Left}\ u)) \\
\equiv&\ ((\mathit{uncurry}\ (\lambda x \rightarrow U) \triangledown \mathit{uncurry}\ (\lambda x \rightarrow V)) \circ \mathit{distl})\ (x, \mathit{Left}\ u) \\
\equiv&\ \mathit{curry}\ ((\mathit{uncurry}\ (\lambda x \rightarrow U) \triangledown \mathit{uncurry}\ (\lambda x \rightarrow V)) \circ \mathit{distl})\ x\ (\mathit{Left}\ u)
\end{aligned}$$

The *Right* v case follows similarly.

9 NON-STANDARD TYPES

So far, we're only handling "standard" types, i.e., primitive types like **1**, *Bool*, *Int*, and *Double*, along with products of standard types and functions between standard types. In practice, most programs also involve algebraic data types. Such "non-standard" types are always isomorphic to standard types. To assist with these isomorphisms, define a class of types with alternative representations. Rather than go all the way to and from standard types in one step, take just a single step at a time:

```
class HasRep a where    -- Law: abst ∘ repr ≡ id
  type Rep a
  repr :: a → Rep a
  abst :: Rep a → a
```

For instance, for a uniform pair type **data** *Pair* a = *P* a a,

```
instance HasRep (Pair a) where
  type Rep (Pair a) = a × a
  repr (P a a′) = (a, a′)
  abst (a, a′) = P a a′
```

The key to translating non-standard types is observing that (a) they are constructed in exactly one way, namely constructor application, and (b) they are consumed in exactly one way, namely as the scrutinee of a **case** expression.

(Haskell's lambda and **let** patterns become simple variable lambda and **let**, together with **case** expressions in GHC Core.) Constructor application becomes *abst* applications, and **case** consumption becomes *repr* applications, both by means of the $abst \circ repr$ law above and some selective inlining.

Given a saturated constructor application $Con\ e_1\ ...\ e_n$, rewrite it to $abst\ (inline\ repr\ (Con\ e_1\ ...\ e_n))$, where $inline\ \ e$ tells GHC's simplifier to inline the expression e.[6] GHC's usual simplifications then eliminate the constructor, leaving only *abst* behind. For example,

$$\begin{aligned}
& P\ 3\ 4 \\
\equiv\ & \{\text{- } abst \circ repr \equiv id \text{ -}\} \\
& abst\ (inline\ repr\ (P\ 3\ 4)) \\
\equiv\ & \{\text{- inline } repr \text{ -}\} \\
& abst\ ((\lambda p \rightarrow \mathbf{case}\ p\ \mathbf{of}\ P\ a\ a' \rightarrow (a, a'))\ (P\ 3\ 4)) \\
\equiv\ & \{\text{- } \beta\text{-reduce -}\} \\
& abst\ (\mathbf{let}\ p = P\ 3\ 4\ \mathbf{in}\ \mathbf{case}\ p\ \mathbf{of}\ P\ a\ a' \rightarrow (a, a')) \\
\equiv\ & \{\text{- } \mathbf{let}\text{-substitute -}\} \\
& abst\ (\mathbf{case}\ P\ 3\ 4\ \mathbf{of}\ P\ a\ a' \rightarrow (a, a')) \\
\equiv\ & \{\text{- GHC's case-of-known-constructor transformation -}\} \\
& abst\ (3, 4)
\end{aligned}$$

Dually, consider a **case** expression $\mathbf{case}\ \ scrut\ \ \mathbf{of}\ \ \{p_1 \rightarrow rhs_1; ...; p_n \rightarrow rhs_n\}$, where (the scrutinee) *scrut* has a non-standard type with a *HasRep* instance. Rewrite *scrut* to $inline\ abst\ (repr\ \ scrut)$ (this time inlining *abst* instead of *repr*). GHC's usual simplifications will then replace the **case** over a non-standard type with a **case** over a standard type or one closer to standard. For instance,

$$\begin{aligned}
& \mathbf{case}\ p\ \mathbf{of}\ P\ x\ y \rightarrow x + y \\
\equiv\ & \{\text{- } abst \circ repr \equiv id \text{ -}\} \\
& \mathbf{case}\ inline\ abst\ (repr\ p)\ \mathbf{of}\ P\ x\ y \rightarrow x + y \\
\equiv\ & \{\text{- inline } abst \text{ -}\} \\
& \mathbf{case}\ (\lambda q \rightarrow \mathbf{case}\ q\ \mathbf{of}\ (a, a') \rightarrow P\ a\ a')\ (repr\ p)\ \mathbf{of}\ P\ x\ y \rightarrow x + y \\
\equiv\ & \{\text{- } \beta\text{-reduce and } \mathbf{let}\text{-substitute -}\} \\
& \mathbf{case}\ (\mathbf{case}\ repr\ p\ \mathbf{of}\ (a, a') \rightarrow P\ a\ a')\ \mathbf{of}\ P\ x\ y \rightarrow x + y \\
\equiv\ & \{\text{- GHC's case-of-case transformation -}\} \\
& \mathbf{case}\ repr\ p\ \mathbf{of}\ (a, a') \rightarrow \mathbf{case}\ P\ a\ a'\ \mathbf{of}\ P\ x\ y \rightarrow x + y \\
\equiv\ & \{\text{- GHC's case-of-known-constructor transformation -}\} \\
& \mathbf{case}\ repr\ p\ \mathbf{of}\ (a, a') \rightarrow \mathbf{let}\ \{x = a; y = a'\}\ \mathbf{in}\ x + y \\
\equiv\ & \{\text{- } \mathbf{let}\text{-substitute -}\} \\
& \mathbf{case}\ repr\ p\ \mathbf{of}\ (a, a') \rightarrow a + a'
\end{aligned}$$

These two transformations occur only in the context "$ccc\ (\lambda x \rightarrow ...)$". The remaining occurrences of *abst* (for construction) and *repr* (for consumption) become part of the categorical vocabulary as a generalization of *HasRep*:

```
class HasRep a ⇒ RepCat k a where        instance HasRep a ⇒ RepCat (→) a where
  reprC :: a ‘k‘ Rep a                      reprC = repr
  abstC :: Rep a ‘k‘ a                      abstC = abst
```

During conversion to categorical form (as in Section 3), *repr* and *abst* are replaced by their generalizations *reprC* and *abstC*. When changing categories (as in Section 4), the occurrences of *reprC* and *abstC* in (→) are replaced by the same methods in another category.

[6] A "saturated" constructor application is one that is applied to the maximal number of arguments, or equivalently, one that has a non-function type. *Unsaturated* constructor applications can be η-expanded until saturated.

Multi-constructor algebraic data types pose no additional difficulty. Their *Rep* encodings involve sums (and often products as well), and GHC's case-of-case and case-of-known-constructor transformations handle the resulting multi-branch **case** expressions. Conversion to categories other than $(\rightarrow)$ requires support for *cocartesian* categories, adding *coproducts*, as well as distributive categories, as described in Section 8.

As an example, the *Graph* category in Section 7.1 handles non-standard types via an additional *Ports* constructor:

```
data Ports :: * → * where
    ...
  AbstP :: Ports (Rep a) → Ports a
```

The *RepCat* methods add and remove *AbstP* ports:

```
instance HasRep a ⇒ RepCat Graph a where
  abstC = Graph (λr → return (AbstP r))
  reprC = Graph (λ(AbstP r) → return r)
```

Many examples with non-standard types compiled to *Graph* appear in (Elliott 2017).

10 SOME IMPLEMENTATION ISSUES

10.1 Unboxed operations

Performance of numeric operations under GHC is considerably improved by using *unboxed* number representations inside of boxed wrappers (Peyton Jones and Launchbury 1991). For instance, the (boxed) *Int* type is defined as a wrapping of an unboxed field:

```
data Int = I Int#
```

Numerical operations on *Int* are then defined to unwrap (unbox) arguments, apply unboxed operations on the contents, and rewrap (rebox) the unboxed results:

```
instance Num Int where
  fromInteger i = I# (integerToInt# i)
  negate (I# x) = I# (negateInt# x)
  I# x + I# y = I# (x +# y)
  I# x − I# y = I# (x −# y)
  I# x ∗ I# y = I# (x ∗# y)
```

When boxed operations are combined, inlined, and optimized, most unwrapping and rewrapping code is eliminated, thanks to the GHC's case-of-known-constructor optimization. For instance, for variables $a, b :: Int$ the expression "$a + 3 * b$" optimizes to the following:

```
case a of I# x → case b of I# y → I# (x +# 3# ∗# y)
```

Even *recursive* definitions involving numbers can be handled efficiently, via the general worker wrapper transformation (Gill and Hutton 2009).

Although unboxing speeds up execution of Haskell programs with the usual interpretation (the $(\rightarrow)$ category), it complicates conversion to categorical form and hence compiling to other categories. Recall the signatures involved in the generalized *Num* class:

```
class NumCat k a where
  addC, subC, mulC :: (a × a) ‘k‘ a
  ...
```

While these methods are polymorphic over number type/object, the polymorphism is implicitly restricted to kind $*$, which means *boxed* types. The original boxed versions of addition and multiplication used in our example above have disappeared from the optimized form and must be recovered in order to satisfy the implicit kind restriction (or some other boxed form, which is probably no easier).

After much experimentation, a simple solution to reboxing emerged. The first step is to find applications of constructors like $I_{\#}$, such as $I_{\#}\ (x +_{\#} 3_{\#} *_{\#} y)$ in the optimized example above. Replace those outer constructors with a synonym defined as follows:

$boxI :: Int_{\#} \rightarrow Int$
$boxI = I_{\#}$
{-# INLINE [0] boxI #-}

The *INLINE* pragma causes these synonyms to be replaced by their original constructors only at the end of compilation (phase zero) if any still exist (which happens with constant expressions). The example above becomes

$\textbf{case}\ a\ \textbf{of}\ I_{\#}\ x \rightarrow \textbf{case}\ b\ \textbf{of}\ I_{\#}\ y \rightarrow boxI\ (x +_{\#} 3_{\#} *_{\#} z)$

The boxing synonyms trigger cascaded application of step-by-step reboxing rewrite rules such as the following:

$$\begin{array}{lll}
\forall u. & boxI\ (negateInt_{\#}\ u) & = negateC\ (boxI\ u) \\
\forall u\ v. & boxI\ (u +_{\#} v) & = addC\ (boxI\ u, boxI\ v) \\
\forall u\ v. & boxI\ (u -_{\#} v) & = subC\ (boxI\ u, boxI\ v) \\
\forall u\ v. & boxI\ (u *_{\#} v) & = mulC\ (boxI\ u, boxI\ v)
\end{array}$$

Introducing category-generalized names rather than the usual versions ((+), (∗), etc) in these rules avoids having GHC's simplifier re-inline the usual versions back into unboxed form. Note here how the reboxing rules push *boxI* applications inward as long as there are unboxed operations being applied. This recursive transformation ends in literals and variables. Our example becomes

$\textbf{case}\ a\ \textbf{of}\ I_{\#}\ x \rightarrow \textbf{case}\ b\ \textbf{of}\ I_{\#}\ y \rightarrow addC\ (boxI\ x, mulC\ (boxI\ 3_{\#}, boxI\ y))$

One more transformation eliminates the unboxing **case** scrutinees: transform an expression like "$\textbf{case}\ a\ \textbf{of}\ I_{\#}\ x \rightarrow \ldots boxI\ x \ldots$" to "$\textbf{let}\ x' = a\ \textbf{in}\ \ldots x' \ldots$". By applying the previous transformations (converting boxing constructors and recursive reboxing) before this unboxing scrutinee elimination, all occurrences of x will be of the form $boxI\ x$, so no unboxed variables remain. The **let** bindings are removed by GHC's simplifier unless doing so replicates nontrivial work. After all of these transformations, our example involves only (a) numeric operations in category-generalized form, (b) variables of boxed types, and (c) boxing synonyms applied to unboxed literals:

$addC\ (a, mulC\ (boxI\ 3_{\#}, b))$

10.2 Translation without closure

Some categories are cartesian but not *cartesian closed*, e.g., vector spaces with linear maps. Most of the rules for converting to CCC form rely on closure, which poses a problem for non-closed categories. If, however, the *overall* function being converted does not involve functions in its domain or codomain, then the corresponding closure-dependent CCC form can often (or perhaps always) be converted to a form free of the *Closed* operations (*apply*, *curry*, and *uncurry*)—assuming that none of the primitive operations (*addC*, *mulC*, *eq*, etc) involve exponentials in their types either (which seems a harmless restriction). An easy way to eliminate the *Closed* operations is by converting first to CCC form in the (→) category (which *is* closed) as suggested in Section 3, applying some rewrite rules that follow from general category laws, and then homomorphically converting to the desired non-closed category, as suggested in Section 4. (For *cartesian closed* categories, compilation can take a more direct route of fusing the vocabulary change with the category change, although homomorphism application is fairly simple and inexpensive.) These closure-eliminating rules include the following:

$$\begin{array}{ll}
\forall f. & uncurry\ (curry\ f) = f \\
\forall g. & curry\ (uncurry\ g) = g \\
\forall g\ f. & apply \circ (curry\ (g \circ exr) \vartriangle f) = g \circ f
\end{array}$$

10.3 Postponing inlining

In the current GHC (8.0.2), class methods are always inlined early. This behavior, if not somehow avoided, would have some unfortunate consequences for compiling to categories as described in this paper:

- The reboxing rewrite rules of Section 10.1 would get undone immediately, through inlining and subsequent simplification, leading to an infinite transformation loop.
- Homomorphism rules (the heart of transforming to non-standard interpretations, outlined in Section 3) would never fire, since they are written in terms of class methods.
- For the same reason, the simplification rules that apply category theory laws would never fire.

Fortunately, there is a simple and effective work-around for premature method inlining. All categorical class methods have corresponding late-inlining aliases with the same names but placed in a dedicated module, along with rewrite rules involving them. Conversion to categorical form (Section 3) uses these aliases instead of the module that defines the classes and methods. If a future version of GHC allows delayed method inlining, the aliases can be removed.

11 RELATED WORK

The Categorical Abstract Machine (CAM) is an execution model for terms from the language of cartesian closed categories (Cousineau et al. 1987), emerging from the categorical combinators of Curien (1986), and used as the basis of an implementation of the Caml dialect and of the ML programming language (Weis et al. 2005). Like the work described in this paper, the CAM is based on CCCs and its relation to the λ-calculus. It does not appear to have been used to give multiple CCC-based interpretations (each with its own semantics and notion of execution).

There has been a lot of work in describing and synthesizing hardware via functional programming, beginning with muFP (Sheeran 1984), which was based on FP (Backus 1978) (close to the language of cartesian categories), extended with streams for synchronous circuits. The muFP language was followed by Ruby, adding a relational perspective (Jones and Sheeran 1990), and then by Lava (Bjesse et al. 1998), which was embedded in Haskell. This very fruitful line of research has focused on describing hardware but also explored elegant expression of hardware-friendly algorithms.

CλaSH is a compiler from Haskell to hardware, also using GHC as a front end and also working by successive program transformation (Baaij and Kuper 2014). Like the work in this paper, CλaSH compiles a somewhat restricted form of Haskell rather than being a Haskell library implemented as a deep or shallow embedding.

Cai et al. (2014) describes how to convert a typed λ-calculus into an incremental version by a process very similar to differentiation. That work informed the incremental CCC of Section 7.6 as another instance of a generalized differentiation CCC, which may have other useful specializations as well (in addition to differential calculus). Also related is the work on self-adjusting computation in a functional setting (Acar 2005; Acar et al. 2005) and the monadic/applicative formulation in Haskell (Carlsson 2002). Both require using a supporting library, with considerable impact of programming style.

Automatic differentiation (AD) has a long and rich history dating back to Wengert (1964) and including modern, functional formulations (Elliott 2009; Karczmarczuk 1998). Siskind and Pearlmutter (2005, 2008) suggest that it is difficult and perhaps impossible to give a correct implementation of AD in purely functional languages such as Haskell, in particular pointing out the danger of "perturbation confusion" when nesting the differentiation operator. The technique in this paper used with the AD CCC given in Section 7.5 side-steps these difficulties by providing a correctly implemented differentiation operator by means of compile-time transformation. It also does not appear to fall naturally into the usual classification of forward vs reverse vs mixed modes, although perhaps it could become any of those modes by applications of the associativity law for composition.

Cartesian categories are closely related to arrows (Hughes 1998), but the latter's inclusion of *arr* (with roughly the the same signature as the *ccc* pseudo-function of Section 5) precludes instances for many cartesian categories.

12 FUTURE WORK

There are several possible improvements to the scheme described in this paper, including the following:

- The categorical classes have fixed versions of product, unit, exponential, coproduct, etc. We can gain much useful flexibility by replacing these fixed type constructions with *associated* types (Chakravarty et al. 2005a,b). For instance, linear maps and polynomials have a "direct sum" coproduct whose representation is the cartesian *product* rather than a sum (tagged union). The types managed by the compiler plugin become somewhat more complicated, but it seems quite doable, and work is in progress.
- The types involved in categorical operations are restricted to having kind $*$. The only reason for this restriction is the fixed versions of product, unit, etc. Once those types are generalized, they can easily become poly-kinded (Hinze 2000b; Yorgey et al. 2012). For instance, the functor versions of linear map representation and operations in Appendix A form a cartesian category, using functor product as the associated categorical product. A related example is the category of natural transformations.
- Implemented as described above, compilation to categories is costly for large computations, with a great deal of inlining, simplification, translation to CCC form, and conversion to alternative categories. When a top-level definition is used more than once, this processing occurs redundantly. Within a given compilation run, perhaps some sort of memoization can be done to reuse work, but the same issue exists between successive compilations and across many modules. An earlier implementation of compiling to categories applied an effective and fairly simple form of separate compilation. For each top-level binding $f\ @v_1\ \ldots\ @v_n = rhs$ (where $v_1, \ldots, v_n$ are type variables), the compiler plugin generated a rewrite rule $ccc\ (f\ @v_1\ \ldots\ @v_n) = ccc\ rhs$. The right-hand side of this rule simplifies to some term rhs', often containing residual *ccc* applications due to polymorphism. Later, when a type instance $ccc\ (f\ @\tau_1 \ldots\ @\tau_n)$ is encountered (for types $\tau_1, \ldots, \tau_n$, possibly containing other type variables), including in a different module, GHC's simplifier would find and apply the generated rule, substituting $\tau_1, \ldots \tau_n$ for $v_1, \ldots v_n$ in rhs', and then continue, making further progress with the unfinished transformation. This experiment in separate compilation worked because the earlier plugin supported only a single CCC, namely *Graph* from Section 7.1. It is not so clear how to adapt this scheme to support multiple categories, including ones not yet defined when a library module is compiled, since translation depends on the instances available for a particular target category.
- In the presence of recursive definitions, repeated inlining and simplification can easily cause the compiler not to terminate. Fortunately, recursion is explicit in GHC Core, so it is easy for a compiler plugin to notice and handle with care. It may thus be possible to translate to a categorical interface for fixed points (Barr 1990; Mulry 1990; Simpson and Plotkin 2000), to then be interpreted in different categories.

Considering the broad applicability of category theory, it seems likely that the applications given in this paper barely skim the surface of the interesting and useful interpretations of functional programs made possible by compiling to categories.

13 CONCLUSIONS

The method described in this paper constitutes a new angle on domain-specific embedded languages (DSELs) and provides a compelling alternative to the "deep embedding" technique often used to enable analysis and optimization (Boulton et al. 1993; Elliott et al. 2003; Gibbons and Wu 2014; Gill 2014). In deep embeddings, a DSEL/library implementation includes a syntactic representation to be manipulated by the library (at its run time), in addition to the host compiler's syntactic representation (GHC Core). Sharing is lost and must be rediscovered by some form of common subexpression elimination (CSE), an awkward and expensive phase to define in a purely functional language like Haskell (Claessen and Sands 1999; Elliott et al. 2003; Gill 2009). The library implementation generally also includes various optimizations on its syntactic representation for the sake of performance, as well as some form of back-end code generation. In these ways, a deep DSEL implementation replicates much of the work of its host compiler, making such implementations difficult, and rarely as high-quality as a host language compiler such as GHC.

In spite of all the effort required, the programming interface of a deep DSEL still has some shortcomings. Instead of manipulating simple values such as *Bool* and *Int* directly, one must operate on some sort of expression type. This fact can be partially hidden by means of overloading, e.g., via instances of the numeric classes.

Some operations, however, have insufficiently flexible types for the required overloading, such as equality and inequalities, as well as **if** … **then** … **else**. Additional efforts can hide more of these symptoms, but the cracks still show, and each new coping mechanism leads to increasingly mysterious type errors. Moreover, deep embeddings cannot support definition-by-cases—a commonly used style in functional programming.

The compiling-to-categories technique described in this paper avoids the shortcomings of shallow and deep embeddings. Unlike shallow embeddings, we can get static analysis—even inside of *functions*—including aggressive optimization. Unlike deep embeddings, one uses Haskell's standard types and notation (directly and without compromise), gets the full power of the host compiler for optimization, and sees the usual type error messages. Moreover, much implementation work is saved. There is no additional syntactic representation to define, and sharing needn't be recovered, since it is never lost.

14 ACKNOWLEDGMENTS

Steve Teig first suggested to me the project of compiling Haskell to (dynamically reprogrammable) hardware, which was the main focus of my work at Tabula. I'm grateful to Steve for inspiration, support, and many conversations during the first phase of this work. Anshul Malvi implemented the graph-to-Verilog translation (also at Tabula). The Kansas University Haskell folks, especially Andrew Farmer and Andy Gill, helped me considerably with their HERMIT system, on which earlier implementations of the Haskell-to-hardware plugin was built (Farmer et al. 2012; Sculthorpe et al. 2013b). Simon Peyton Jones suggested structuring the compiler plugin as a "built-in" GHC rewrite rule. Tim Sears has provided helpful discussions, encouragement, and suggestions throughout the project. Finally, my thanks to Target, for supporting continued development.

REFERENCES

Umut Acar. *Self-adjusting computation.* PhD thesis, School of Computer Science, Carnegie Mellon University, May 2005.

Umut A. Acar, Guy E. Blelloch, Matthias Blume, and Robert Harper. Self-adjusting programming. In *ML Workshop*, 2005.

Steve Awodey. *Category theory*, volume 49 of *Oxford Logic Guides*. Oxford University Press, 2006.

Christiaan Baaij and Jan Kuper. Using rewriting to synthesize functional languages to digital circuits. In *Trends in Functional Programming, Lecture Notes in Computer Science*, pages 17–33, 2014.

John Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

Michael Barr. Fixed points in cartesian closed categories. *Theoretical Computer Science*, 70(1):65–72, 1990.

Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: hardware design in Haskell. In *ICFP*, pages 174–184, 1998.

Guy E. Blelloch. Prefix Sums and Their Applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, November 1990.

Max Bolingbroke. Constraint kinds for GHC. http://blog.omega-prime.co.uk/?p=127, 2011.

Richard Boulton, Andrew Gordon, Mike Gordon, John Harrison, John Herbert, and John Van Tassel. Experience with embedding hardware description languages in HOL. In *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, volume A-10, pages 129–156, 1993.

Yufei Cai, Paolo G. Giarrusso, Tillmann Rendel, and Klaus Ostermann. A theory of changes for higher-order languages: incrementalizing λ-calculi by static differentiation. In *PLDI '14*, pages 145–155, 2014.

Magnus Carlsson. Monads for incremental computing. In *ICFP*, pages 26–35, 2002.

Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. In *ICFP*, pages 241–253, 2005a.

Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. In *POPL*, pages 1–13, 2005b.

Mark Chu-Carroll. Interpreting lambda calculus using closed cartesian categories, March 2012. URL http://goodmath.scientopia.org/2012/03/11/interpreting-lambda-calculus-using-closed-cartesian-categories/.

Koen Claessen and David Sands. Observable Sharing for Functional Circuit Description. In *Asian Computing Science Conference*, 1999.

Guy Cousineau, Pierre-Louis Curien, and Michel Mauny. The Categorical Abstract Machine. *Science of Computer Programming*, 8, 1987.

Pierre-Louis Curien. Categorical combinators. *Information and Control*, 69(1-3):188–254, 1986.

Conal Elliott. Beautiful differentiation. In *International Conference on Functional Programming (ICFP)*, 2009.

Conal Elliott. Generic functional parallel algorithms: Scan and FFT. Submitted for publication to ICFP 2017, February 2017.

Conal Elliott, Sigbjørn Finne, and Oege de Moor. Compiling embedded languages. *Journal of Functional Programming*, 13(2), 2003.

Andrew Farmer, Andy Gill, Ed Komp, and Neil Sculthorpe. The HERMIT in the machine: A plugin for the interactive transformation of GHC core language programs. In *Proceedings of the ACM SIGPLAN Haskell Symposium*, Haskell '12, pages 1–12. ACM, 2012. ISBN 978-1-4503-1574-6. doi: 10.1145/2364506.2364508.

Jeremy Gibbons. Calculating functional programs. In Roland Backhouse, Roy Crole, and Jeremy Gibbons, editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, volume 2297 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

Jeremy Gibbons and Nicolas Wu. Folding domain-specific languages: deep and shallow embeddings (functional pearl). In *ICFP*, 2014.

Andy Gill. Type-safe observable sharing in Haskell. In *Haskell Symposium*, pages 117–128, September 2009.

Andy Gill. Domain-specific languages and code synthesis using Haskell. *ACM Queue*, 12(4), April 2014.

Andy Gill and Graham Hutton. The worker/wrapper transformation. *Journal of Functional Programming*, pages 227–251, 2009.

Ralf Hinze. Memo functions, polytypically! In *Proceedings of the 2nd Workshop on Generic Programming*, pages 17–32, 2000a.

Ralf Hinze. Polytypic values possess polykinded types. In *Science of Computer Programming*, pages 2–27. Springer-Verlag, 2000b.

Ralf Hinze. Fun with phantom types. In *The fun of programming*. Palgrave, 2003.

John Hughes. The design of a pretty-printing library. In *Advanced Functional Programming*, pages 53–96. Springer Verlag, 1995.

John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, 1998.

John Hughes and Simon Peyton Jones. The *pretty* package. https://hackage.haskell.org/package/pretty, November 2007. Haskell library.

Geraint Jones and Mary Sheeran. Circuit design in Ruby. *Formal methods for VLSI design*, 1, 1990.

Mark P. Jones. Dictionary-free overloading by partial evaluation. In *In ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 107–117, 1994.

Jerzy Karczmarczuk. Functional differentiation of computer programs. In *ICFP*, pages 195–203, New York, NY, USA, 1998. ACM.

Edward Kmett. What constraints entail: Part 1, November 2011. URL http://comonad.com/reader/2011/what-constraints-entail-part-1/.

Joachim Lambek. From λ-calculus to cartesian closed categories. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Academic Press, 1980.

Joachim Lambek. Cartesian closed categories and typed lambda-calculi. In *Proceedings of the Thirteenth Spring School of the LITP on Combinators and Functional Programming Languages*, pages 136–175, 1986.

F. William Lawvere and Stephen H. Schanuel. *Conceptual Mathematics: A First Introduction to Categories*. Cambridge University Press, 2nd edition, 2009.

Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *Conference on Domain-Specific Languages*, pages 109–122, 1999.

José Pedro Magalhães, Atze Dijkstra, Johan Jeuring, and Andres Löh. A generic deriving mechanism for Haskell. In *Proceedings of the Haskell Symposium*, pages 37–48, 2010.

José Pedro Magalhães et al. GHC.Generics, 2011. URL https://wiki.haskell.org/GHC.Generics. Haskell wiki page.

M. Douglas McIlroy. Power series, power serious. *Journal of Functional Programming*, 9(3):325–337, 1999.

R.E. Moore. *Interval analysis*. Series in automatic computation. Prentice-Hall, 1966.

Philip S. Mulry. Categorical fixed point semantics. *Theoretical Computer Science*, 70(1):85–97, January 1990.

Simon Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Functional programming languages and computer architecture*, pages 636–666, 1991.

Simon Peyton Jones and Simon Marlow. Secrets of the Glasgow Haskell compiler inliner. *Journal of Functional Programming*, 12(5), July 2002.

Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Haskell Workshop*, pages 203–233, 2001.

Neil Sculthorpe, Jan Bracker, George Giorgidze, and Andy Gill. The constrained-monad problem. In *ICFP*, pages 287–298, 2013a.

Neil Sculthorpe, Andrew Farmer, and Andy Gill. The HERMIT in the tree: Mechanizing program transformations in the GHC core language. In *Proceedings of the 24th Symposium on Implementation and Application of Functional Languages*, volume 8241 of *Lecture Notes in Computer Science*, pages 86–103, 2013b.

Mary Sheeran. muFP, a language for VLSI design. In *ACM Symposium on LISP and Functional Programming*, pages 104–112, 1984.

Alex Simpson and Gordon Plotkin. Complete axioms for categorical fixed-point operators. In *Logic in Computer Science*, pages 30–41, 2000.

Jeffrey Mark Siskind and Barak A. Pearlmutter. Perturbation confusion and referential transparency: correct functional implementation of forward-mode AD. In *Implementation and Application of Functional Languages*, pages 1–9, September 2005.

Jeffrey Mark Siskind and Barak A. Pearlmutter. Nesting forward-mode AD in a functional framework. *Higher Order Symbolic Computation*, 21 (4):361–376, 2008.

Michael Spivak. *Calculus on Manifolds: A Modern Approach to Classical Theorems of Advanced Calculus*. HarperCollins Publishers, 1971.

Martin Sulzmann, Manuel M. T. Chakravarty, Simon L. Peyton Jones, and Kevin Donnelly. System F with type equality coercions. In *TLDI*, pages 53–66, 2007.

Pierre Weis et al. A history of Caml, 2005. URL https://caml.inria.fr/about/history.en.html. Last updated 2005-01-28.

R. E. Wengert. A simple automatic derivative evaluation program. *Communications of the ACM*, 7(8):463–464, 1964.

Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon L. Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving Haskell a promotion. In *TLDI*, pages 53–66. ACM, 2012.

A LINEAR MAP DETAILS

The linear map category in Section 7.4 is based on free vector spaces (or modules or semimodules) over a scalar s, as representable functors, i.e., functors isomorphic to $A \to s$ for some type A. For instance, the vector space $\mathbb{R}^2$

over ℝ is represented as a *Pair* ℝ, where *Pair* is a uniform pair functor, which is isomorphic to *Bool* → ℝ. This representation enables simple and general definitions of "vector" operations, e.g.,

```
scaleV :: (Functor f, Num s) ⇒ s → f s → f s          -- Scale a vector
s `scaleV` v = (s ∗) <$> v

addV :: (Zip f, Num s) ⇒ f s → f s → f s               -- Add vectors
addV = zipWith (+)

dotV :: (Zip f, Foldable f, Num s) ⇒ f s → f s → s     -- Dot product
x `dotV` y = sum (zipWith (∗) x y)

zeroV :: (Pointed f, Num a) ⇒ f a                       -- Zero vector
zeroV = point 0
```

The *Pointed* class has one method *point* :: $s \to f\ s$ that fills a structure with a given value. The *Zip* class provides *zipWith* :: $(s \to t \to u) \to f\ s \to f\ t \to f\ u$ for combining structures element-wise. These operations can be generalized easily and usefully from *Num* to semirings.

A functor version of linear maps from $a\ s$ to $b\ s$ in row-major form (though column-major can work as well):

```
infixr 1 ⊸
type (a ⊸ b) s = b (a s)
```

To apply a linear map *as* :: $b\ (a\ s)$ to a vector $a :: a\ s$, form the inner product of each row in *as* with a:

```
lapplyL :: (Zip a, Foldable a, Zip b, Num s) ⇒ (a ⊸ b) s → a s → b s
lapplyL as a = (`dotV` a) <$> as
```

The identity and composition for this linear map representation are fairly simple:

```
idL :: (Diagonal a, Num s) ⇒ (a ⊸ a) s
idL = diag 0 1

compL :: (Zip a, Zip b, Pointed a, Foldable b, Functor c, Num s) ⇒ (b ⊸ c) s → (a ⊸ b) s → (a ⊸ c) s
bc `compL` ab = (λb → sumV (zipWith scaleV b ab)) <$> bc
```

The *Diagonal* class has *diag* :: $s \to s \to f\ (f\ s)$, with *diag z o* having o ("one") on the diagonal and z ("zero") elsewhere. While *idL* and *compL* are used in the *Category* instance in Section 7.4, the following three are used in the *Cartesian* instance:

```
exlL :: (Pointed a, Diagonal a, Pointed b, Num s) ⇒ (a × b ⊸ a) s
exlL = (× zeroV) <$> idL

exrL :: (Pointed b, Diagonal b, Pointed a, Num s) ⇒ (a × b ⊸ b) s
exrL = (zeroV ×) <$> idL

forkL :: (a ⊸ b) s → (a ⊸ c) s → (a ⊸ b × c) s
forkL = (×)

itL :: (a ⊸ U₁) s
itL = U₁
```

There are dual operations for the categorical coproduct, which are represented as cartesian products:[7]

```
inlL :: (Pointed a, Diagonal a, Pointed b, Num s) ⇒ (a ⊸ a × b) s
inlL = idL × zeroL
```

[7]Using these definitions to give a *Cocartesian* instance requires an instance-associated type for coproducts. This generalization and poly-kinded classes (both discussed in Section 12) enable the functor representation of linear maps to have *Category*, *Cartesian*, and *Cocartesian* instances as well. Linear maps are not distributive or cartesian closed, however.

$inrL :: (Pointed\ a, Pointed\ b, Diagonal\ b, Num\ s) \Rightarrow (b \multimapdot a \times b)\ s$
$inrL = zeroL \times idL$

$joinL :: Zip\ c \Rightarrow (a \multimapdot c)\ s \to (b \multimapdot c)\ s \to (a \times b \multimapdot c)\ s$
$joinL = zipWith\ (\times)$

Finally, the function *linearL* that converts from a function (presumably linear) to the functor representation of linear maps:

class $OkLF\ f \Rightarrow HasL\ f$ **where**
  $linearL :: \forall s\ g.(Num\ s, OkLF\ g) \Rightarrow (f\ s \to g\ s) \to (f \multimapdot g)\ s$ -- Law: $linearL \circ lapplyL \equiv id$

instance $HasL\ U_1$ **where** $linearL\ h = const\ U_1 \mathbin{<\$>} h\ U_1$

instance $HasL\ Par_1$ **where** $linearL\ h = Par_1 \mathbin{<\$>} h\ (Par_1\ 1)$

instance $(HasL\ f, HasL\ g) \Rightarrow HasL\ (f \times g)$ **where**
  $linearL\ h = linearL\ (h \circ (\times\ zeroV))\ `joinL`\ linearL\ (h \circ (zeroV\ \times))$

The constraint used in defining $Ok\ (\multimap_s)$ conjoins required class constraints:

type $OkLF\ a = (Foldable\ a, Pointed\ a, Zip\ a, Diagonal\ a)$