Compiling to Categories

CONAL ELLIOTT, Target, USA

It is well-known that the simply typed lambda-calculus is modeled by any cartesian closed category (CCC). This correspondence suggests giving typed functional programs a variety of interpretations, each corresponding to a different category. A convenient way to realize this idea is as a collection of meaning-preserving transformations added to an existing compiler, such as GHC for Haskell. This paper describes such an implementation and demonstrates its use for a variety of interpretations including hardware circuits, automatic differentiation, incremental computation, and interval analysis. Each such interpretation is a category easily defined in Haskell (outside of the compiler). The general technique appears to provide a compelling alternative to deeply embedded domain-specific languages.

CCS Concepts: • Theory of computation → Lambda calculus; • Software and its engineering → Functional languages; Compilers;

Additional Key Words and Phrases: category theory, compile-time optimization, domain-specific languages

ACM Reference Format:
https://doi.org/10.1145/3110271

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 nth 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 any 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 [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:

```haskell
infixr 9 ♂
class Category k where
  id :: a ♂ a
  (○) :: (b ♂ k) c → (a ♂ k) b → (a ♂ k) c

instance Category (→) where
  id = λx → x
  g ○ f = λx → g (f x)
```

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

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

```haskell
infixr 3 △
class Cartesian k where
  (△) :: (a ♂ k' c) → (a ♂ k' (c ⋉ d))
  exl :: (a × b) ♂ k' a
  exr :: (a × b) ⿂ k' b

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

In this paper, “a × 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:

\[ ∀h. \ h \equiv f \triangle g \iff \textrm{exl } h \equiv f \land \textrm{exr } h \equiv g \]

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 1 (corresponding to Haskell’s () type), such that there is exactly one k-arrow from a to 1 for any object a in k:

```haskell
class Category k ⇒ Terminal k where
  it :: a ⿂ 1

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

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

```haskell
class Cartesian k ⇒ Closed k where
  apply :: ((a ⇒ b) × a) ⿂ k' b
  curry :: ((a × b) ⿂ k' c) → (a ⿂ k' (b ⇒ c))
  uncurry :: (a ⿂ k' (b ⇒ c)) → ((a × b) ⿂ 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:

\[ apply \circ (curry f \circ \textrm{exl } \triangle \textrm{exr }) \equiv f \]

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 Bools. 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. A translation from the STLC can be defined in terms of typing contexts [Chu-Carroll 2012; Curien 1986]. In the translation as described below, however, every translated term is instead a closed, explicit λ-abstraction—an especially convenient style for use from within GHC’s simplifier (Section 5), which 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, \( \lambda x : \tau \to U \) for some term \( U \); otherwise, simply \( \eta \)-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 variable choice, so we must have the identity function on \( \tau \):

\[
(\lambda x : \tau \to x) \equiv id : \tau \to \tau.
\]

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

\[
\begin{align*}
\lambda x &\to U V \equiv \{ \text{definition of apply on (→)} \}\ \\lambda x \to apply (U, V) \\
\lambda x &\to (\lambda x \to (\lambda x \to U \Delta (\lambda x \to V)) x) \equiv \{ \text{definition of (△) on (→)} \}\ \\lambda x \to apply \circ ((\lambda x \to U) \Delta (\lambda x \to V))
\end{align*}
\]
If the body of an abstraction is an abstraction, we can curry a translation of the uncurried form:

\[
\lambda x \to \lambda y \to U
\]

\[
\equiv \{ \text{definition of curry on (\to)} \} \]

\[
\text{curry } (\lambda (x, y) \to U)
\]

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

\[
(\lambda x \to \text{case scrut of } \{ (u, v) \to rhs \}) \equiv (\lambda x \to \text{let } \{ w = \text{scrut}; u = \text{exl } z; v = \text{exr } z \} \text{ in rhs})
\]

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., \(\lambda x \to c\). There are two possibilities.

If \(c\) has function type and an interpretation via \(\text{BoolCat}, \text{NumCat}\), etc, translate it accordingly:

\[
(\lambda x \to \neg) \equiv \text{constFun \ notC} \\
(\lambda x \to (\land)) \equiv \text{constFun (\text{curry andC})}
\]

...where

\[
\text{constFun :: Closed } k \Rightarrow (a \ 'k' \ b) \to (z \ 'k' \ (a \Rightarrow b))
\]

\[
\text{constFun } f = \text{curry } (f \circ \text{exr})
\]

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

\[
sqr :: \text{Num } a \Rightarrow a \to a
\]

\[
sqr a = a \ast a
\]

\[
\text{magSqr} :: \text{Num } a \Rightarrow a \times a \to a
\]

\[
\text{magSqr } (a, b) = \text{sqr } a + \text{sqr } b
\]

\[
\text{cosSinProd} :: \text{Floating } a \Rightarrow a \times a \to a \times a
\]

\[
\text{cosSinProd } (x, y) = (\text{cos } z, \text{sin } z) \quad \text{where } z = x \ast y
\]

Type-specialized to a primitive type like \(\text{Double}\), and converted to CCC form:

\[
\text{sqr} = \text{mulC } \circ (\text{id } \triangle \text{id})
\]

\[
\text{magSqr} = \text{addC } \circ (\text{mulC } \circ (\text{exl } \triangle \text{exl}) \triangle \text{mulC } \circ (\text{exr } \triangle \text{exr}))
\]

\[
\text{cosSinProd} = (\text{cosC } \triangle \text{sinC}) \circ \text{mulC}
\]

The \(\text{sqr}\) conversion, step-by-step:

\[
\text{sqr } @\text{Double}
\]

\[
\equiv \{ \text{inlining and simplifications by GHC} \}
\]

\[
\lambda x \to \text{timesDouble } x \ x
\]

\[
\equiv \{ \text{translate application} \}
\]

\[
\text{apply } \circ ((\lambda x \to \text{timesDouble } x) \triangle (\lambda x \to x))
\]

\[
\equiv \{ \text{\(\eta\) reduction by GHC} \}
\]

\[
\text{apply } \circ (\text{timesDouble } \triangle (\lambda x \to x))
\]

\[
\equiv \{ \text{translate \(\text{timesDouble}\) and variable} \}
\]

\[
\text{apply } \circ (\text{curry \text{mulC } \triangle \text{id}})
\]
\[
\begin{align*}
    \mathcal{H} \text{id} & \equiv \text{id} \\
    \mathcal{H} (g \circ f) & \equiv \mathcal{H} g \circ \mathcal{H} f \\
    \mathcal{H} (f \triangle g) & \equiv \mathcal{H} f \triangle \mathcal{H} g \\
    \mathcal{H} \text{exl} & \equiv \text{exl} \\
    \mathcal{H} \text{exr} & \equiv \text{exr} \\
    \mathcal{H} (\text{apply } f) & \equiv \text{apply } (\mathcal{H} f) \\
    \mathcal{H} (\text{uncurry } g) & \equiv \text{uncurry } (\mathcal{H} g)
\end{align*}
\]

(a) Category  (b) Cartesian  (c) Closed

Fig. 1. Homomorphism properties

≡ { Law for closed categories }

\[\text{mulC} \circ (\text{id} \triangle \text{id})\]

That last law, \(\text{apply} \circ (\text{curry } h \triangle g) \equiv h \circ (\text{id} \triangle g)\), 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, \(\text{cosSinProd}\) becomes \((\text{cosC} \triangle \text{sinC}) \circ \text{mulC} \circ (\text{exl} \triangle \text{exr})\).

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 \(\text{Category}, \text{Cartesian},\) and \(\text{Closed}\). Note that 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 \((\to)\)) 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 \(\lambda\)-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

\[\text{The properties are also known as “functor”, “cartesian functor”, and “closed cartesian functor”, respectively. Note that Haskell’s standard library comes with a \textit{Functor} type class, but it is restricted to endofunctors on the standard Haskell category (i.e., from \((\to)\) to \((\to)\)).}\]
monomorphic, then the intermediate polymorphism can be removed by monomorphizing, i.e.,
lining 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, similarly to the treatment by Jones [1994].

In practice, the transformation to CCC form and conversion from \((\to)\) to other CCCs are
implemented as GHC rewrite rules [Peyton Jones et al. 2001]. Transformation is guided by the
presence of a pseudo-function:\footnote{More generally, \texttt{ccc} would have the signature of a categorical
functor, i.e., \((a \to b) \to (f \cdot a' \cdot k' \cdot f \cdot b)\). This additional flexibility
complicates translation, but is quite useful (as alluded to in Section 12) and will be addressed in a later paper.}

\[
\text{ccc} :: (a \to b) \to (a \cdot k \cdot b)
\]

One might expect there to be a constraint on \(k\), such as \texttt{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. Homomor-
phism rules similar to the \(H\) examples of Section 2 push applications of \texttt{ccc} inward, eventually
disappearing, as in the rules for \texttt{id}, \texttt{exl}, \texttt{exr}, and \texttt{apply} above as well as the similar rules for primitives
like \texttt{addC} and \texttt{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 \texttt{ccc}-specific
transformation and more general transformations helps 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 \texttt{CoreExpr} \(\to\) \texttt{Maybe CoreExpr},
injected into GHC’s simplifier [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.

The argument to \texttt{ccc} is typically a lambda expression, although it might need to be converted to
one via \(\eta\)-expansion or inlining. In such cases, the \texttt{ccc} rule will perform one transformation step
described in Sections 3 and 4.

For instance, the currying transformation from Section 3, for a target category \(k\), is

\[
\text{ccc} \left( \lambda x \to U \right. V \left. \right) \quad \longmapsto \quad \text{apply}_k \circ \left( \text{ccc} \left( \lambda x \to U \right) \cdot \text{ccc} \left( \lambda x \to V \right) \right)
\]

where the \(k\) subscripts here indicate type applications of the categorical methods to the target
category \(k\). Since Core lacks the type classes and methods found in the Haskell source language,
the transformation rule must construct instances explicitly and insert the \(k\)-specific versions of the
general methods. These instances are in the form of “class dictionaries”, which are simply records
of methods (Core value identifiers). GHC performs many other simplifications, including method
name inlining. The generated \texttt{ccc} calls, if any, are then further transformed by future applications
of the \texttt{ccc} transformation. Some transformations introduce no new \texttt{ccc} calls, instead generating
simple expressions like \texttt{id}_k and \texttt{exl}_k.

Adding a new built-in rule to GHC requires writing a Core-to-Core plugin [GHC Team 2016],
packaging some code that alters the normal compiler flow. While plugins can be arbitrarily complex,
very simple one suffices, attaching one the transformation function to the \texttt{ccc} identifier for GHC’s
Core simplifier to find and apply during normal compilation. Most of the implementation work is
in this transformation function, which takes a Core expression \(e\) and successfully yields an altered expression \(\text{Just } e'\) or fails \(\text{Nothing}\) based on the structure of \(e\), as described above.

6 CONSTRUED 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 for constrained monads, leading to a normal form for applicative and monadic computations that allowed use of the usual, unconstrained Monad class [Sculthorpe et al. 2013a]. 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:

```haskell
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 \(◦\):

```haskell
  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.\(^3\)

```haskell
  data Graph a b = Graph (Ports a → GraphM (Ports b))
```

\(^3\)We could instead combine State Port with Writer [Comp], but the form shown more easily extends optimizations discussed below.
type GraphM = State (Port, [Comp])
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:

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

The "<<<" operator is Kleisli composition, of type Monad m ⇒ (b → m c) → (a → m b) → (a → m c).

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:

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

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. 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 are 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 @Int))

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

One can convert graphs to other forms as well. For instance, it was easy to translate to shader programs for parallel execution on graphics processors, as in Vertigo [Elliott 2004], and to SMT (satisfiability modulo theories) problems to be solved by Z3 [De Moura and Bjørner 2008].

7.2 Syntax

Section 3 showed CCC expressions for three simple functions \((sqr, magSqr, \text{and cosSinProd})\). 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:

\[
\text{type } \text{DocTree} = \text{Tree } PDoc
\]

where \(\text{Tree } a\) 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):

\[
\text{data } \text{Tree } a = \text{Node } a \ [\text{Tree } a]
\]

\[
\text{type } PDoc = \text{Rational } \rightarrow \text{Doc}
\]

One could use \(\text{String}\) in place of \(PDoc\) in the definition of \(\text{DocTree}\), 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:

\[
\text{prettyTree :: DocTree } \rightarrow \ PDoc
\]

\[
\text{prettyTree (Node d [u, v]) } p \mid \text{Just } (q, (lf, rf)) \leftarrow \text{lookup name fixities} =
\]

\[
\text{maybeParens } (p > q) \$ \text{sep } [\text{prettyTree u (lf q) } \leftrightarrow \text{text name, (prettyTree v) (rf q)}]
\]

\[
\text{where name } = \text{show } (d 0)
\]

\[
\text{fixities} = \text{fromList } [("ο", (9, assocR)), ("△", (3, assocR)), ("▽", (2, assocR))]
\]

\[
\text{prettyTree (Node f es) } p =
\]

\[
\text{maybeParens } (\neg (\text{null es}) \land p > 10) \ (\text{sep } (f 10 : \text{map } (λe \rightarrow \text{prettyTree e} 11) \ es))
\]

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)

assocL, assocR, nonassoc :: Assoc

assocL = (id, succ)
assocR = (succ, id)
nonassoc = (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:

\[
\text{newtype } \text{Syn } a \ b = \text{Syn } \text{DocTree}
\]

\[
\text{atom } :: \text{Pretty } a \Rightarrow a \rightarrow \text{Syn } a \ b
\]

\[
\text{atom } a = \text{Syn } (\text{Node } (\text{ppretty } a) \ [])
\]

\[
\text{appt } :: \text{String } \rightarrow [\text{DocTree}] \rightarrow \text{DocTree}
\]

\[
\text{appt } s \ ts = \text{Node } (\text{const } (\text{text } s)) \ ts
\]

\[
\text{app } 0 :: \text{String } \rightarrow \text{Syn } a \ b
\]

\[
\text{app } 0 \ s = \text{Syn } (\text{appt } s \ [])
\]

\[
\text{app } 1 :: \text{String } \rightarrow \text{Syn } a \ b \rightarrow \text{Syn } c \ d
\]

\[
\text{app } 1 \ s (\text{Syn } p) = \text{Syn } (\text{appt } s \ [p])
\]

\[
\text{app } 2 :: \text{String } \rightarrow \text{Syn } a \ b \rightarrow \text{Syn } c \ d \rightarrow \text{Syn } e \ f
\]

\[
\text{app } 2 \ s (\text{Syn } p) (\text{Syn } q) = \text{Syn } (\text{appt } s \ [p, q])
\]

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

\[
\text{instance } \text{Category } \text{Syn } \text{where}
\]

\[
\text{id } = \text{app } 0 \ "\text{id}"
\]

\[
(\circ) = \text{app } 2 \ "\text{o}"
\]

\[
\text{instance } \text{Cartesian } \text{Syn } \text{where}
\]

\[
\text{exl } = \text{app } 0 \ "\text{exl}"
\]

\[
\text{exr } = \text{app } 0 \ "\text{exr}"
\]

\[
(\triangle) = \text{app } 2 \ "\triangle"
\]

\[
\text{instance } \text{Closed } \text{Syn } \text{where}
\]

\[
\text{apply } = \text{app } 0 \ "\text{apply}"
\]

\[
\text{curry } = \text{app } 1 \ "\text{curry}"
\]

\[
\text{uncurry } = \text{app } 1 \ "\text{uncurry}"
\]

\[
\text{instance } \text{BoolCat } \text{Syn } \text{where}
\]

\[
\text{andC } = \text{app } 0 \ "\text{andC}"
\]

\[
\text{orC } = \text{app } 0 \ "\text{orC}"
\]

\[
\ldots
\]

These “instances”, while useful for debugging, fail to satisfy the categorical axioms.

### 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? We could compile twice, each with a different target category, but 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:

\[
\text{infixl } 7 \otimes
\]

\[
\text{data } (p \otimes q) \ a \ b = p \ a \ b \otimes q \ a \ b
\]

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

\[
\text{instance } (\text{Category } k, \text{Category } k') \Rightarrow \text{Category } (k \otimes k') \text{ where}
\]

\[
\text{type } \text{Ok } (k \otimes k') \ a = (\text{Ok } k \ a, \text{Ok } k' \ a)
\]

\[
\text{id } = \text{id } \otimes \text{id}
\]

\[
(\text{g } \otimes \text{g'})(\text{f } \otimes \text{f'}) = (\text{g } \circ \text{f})(\text{g'} \circ \text{f'})
\]

\[
\text{instance } (\text{Cartesian } k, \text{Cartesian } k') \Rightarrow \text{Cartesian } (k \otimes k') \text{ where}
\]

\[
\text{exl } = \text{exl } \otimes \text{exl}
\]

\[
\text{exr } = \text{exr } \otimes \text{exr}
\]

\[
(\text{f } \otimes \text{f'})(\text{\triangle}(\text{g } \otimes \text{g'})) = (\text{f } \triangle \text{g})(\text{f'} \triangle \text{g'})
\]

\[
\ldots
\]
As an identity for \((\otimes)\), there is a category with exactly one arrow for each domain/codomain pair and trivial instances of Category, Cartesian, etc.

### 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 \(f :: A \to s\) from some index set \(A\). Rather than mapping to functions, however, we can use a memoized form for these functions as tries, composed from some basic functor building blocks [Hinze 2000].

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

```haskell
class HasV s a where
    type V s a :: * \to *  -- "vector form"
    toV :: a \to V s a s
    unV :: V s a s \to a
```

Some instances:

```haskell
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) \Rightarrow HasV s (a \times b) where
    type V s (a \times b) = V s a \times V s b
    toV (a, b) = toV a \times toV b
    unV (f \times g) = (unV f, unV g)
```

The \(\text{Par}_1\), \(\text{U}_1\) and \((\times)\) type constructors are identity functor, unit functor, and cartesian functor product, taken from GHC.Generics [Magalhães et al. 2010; Magalhães et al. 2011]:

```haskell
data U1 s = U1

data Par1 s = Par1 s

data (f \times g) s = f g \times 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):

```haskell
newtype a \o s b = LMap (V s b (V s a s))
```

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

```haskell
instance Num s \Rightarrow Category (\o_s) where
    type Ok (\o_s) a = (HasV s a, OkLF (V s a))
```

id = LMap idL
LMap g \circ LMap f = LMap (g \textit{`compL`} f)

\textbf{instance Num s ⇒ Cartesian \((-\circ,\)\) where}
\text{exl} = LMap exlL
\text{exr} = LMap exrL
LMap g \triangle LMap f = LMap (g \textit{`forkL`} f)

\textbf{instance Num s ⇒ Terminal \((-\circ,\)\) where}
it = LMap UL

Linear map \textit{application} is a functor from \((-\circ,\) to \(\rightarrow\), serving as a semantic function for \((-\circ,\)):

\textit{lapply} :: (Ok_2 \(-\circ,\) a b, Num s) ⇒ (a \rightarrow b) \rightarrow (a \rightarrow b)
\textit{lapply} (LMap ba) = unV \circ lapplyL ba \circ toV

Conversely, given a function \(f :: a \rightarrow b\), we can construct a linear map \(f' :: a \rightarrow (\circ, b)\) such that \textit{lapply} \(f' \equiv f\) if \(f\) is linear:

\textit{linear} :: (Ok_2 \(-\circ,\) a b, HasL (V s a), Num s) ⇒ (a \rightarrow b) \rightarrow (a \rightarrow (\circ, b))
linear h = LMap (linearL (toV \circ h \circ unV))

\section{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:

\textit{deriv} :: (a \rightarrow b) → (a \rightarrow (a \rightarrow (\circ, b)))

Although \textit{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]:

\textit{deriv} \((g \circ f)\) \(a\) ≡ \textit{deriv} \(g\) \((f\ a)\) \circ \textit{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 \textit{deriv} is not a functor. All is not lost though, as we can instead compositionally construct a combination of functions \textit{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} \(a \rightarrow\circ s\) \(b = D\ (a \rightarrow b)\ (a \rightarrow (a \rightarrow s))\) \hspace{1em} -- 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 \((\triangle)\) isomorphism from \textit{Cartesian} to combine functions and their derivatives:

\textbf{data} \(a \rightarrow\circ s\) \(b = D\ \{\text{unD} :: a \rightarrow b \times (a \rightarrow s)\}\) \hspace{1em} -- allows work sharing

Our goal is to implement the following functor
andDeriv :: (a → b) → (a ∼₅ b)
andDeriv f = D (f ∆ deriv f) -- specification

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

deriv f = snd \ o \ unD (andDeriv f)

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

linearD :: (Num s, Ok₂ (−₀₅) a b) ⇒ (a → b) → (a −₀₅ b) → (a ∼₅ b)
linearD f f’ = D (f ∆ const f’)

Now we’re ready to define the Category instance for differentiable functions. The identity function is linear, and composition follows from the chain rule:

instance Num s ⇒ Category (∼₅₁) where
  type Ok (∼₅₁) a = Ok (−₀₅) a
  id = linearD id id
  D g ∘ D f = D (λa → let \{(b, f’) = f a; (c, g’) = g b\} in \{(c, g’ ∘ f’\})

Product operations are handled similarly. For (∆), we’ll need a counterpart to the chain rule:

deriv (f ∆ g) a = deriv f a ∆ deriv g a

Assembling the pieces,

instance Num s ⇒ Cartesian (∼₅₁) where
  exl = linearD exl exl
  expr = linearD expr expr
  D f ∆ D g = D (λa → let \{(b, f’) = f a; (c, g’) = g a\} in \{(b, c), f’ ∆ g’\})

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

instance (Num s, V s s ∼ Par₁, Ok (−₀₅) s) ⇒ NumCat (∼₅₁) s where
  negateC = linearD negateC (linear negateC)
  addC = linearD addC (linear addC)
  mulC = D (mulC ∆ λ(a, b) → linear (λ(da, db) → da ∗ b + db ∗ a))

...

Figures 4 and 5 shows the result of andDeriv on the magSqr and cosSinProd examples from Section 3. Because magSqr :: ℝ² → ℝ, its derivative values have type ℝ² → ℝ, represented by a pair of reals. Because cosSinProd :: ℝ² → ℝ², its derivative values have type ℝ² → ℝ², represented by a pair of pairs of reals. Figures 6 and 7 shows the results of deriv on the same two examples.

The Category and Cartesian instances for (∼₅₁) rely on (−₀₅) only for its instances of these same classes. Thanks to this simple relationship, we can easily generalize from (−₀₅) to an arbitrary bicartesian category, re-specializing to (∼₅₁):

newtype GD k a b = D (a → b × (a ‘k’ b))

type (∼₅₁) = GD (−₀₅)

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

---

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.
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, define an interface for incremental value changes, mimicking Cai et al. [2014]:

\[
\text{infixl} 6 \triangleleft, \triangleright
\]

class HasDelta a where

- type \( \Delta a \)
- \((\triangleleft) \) :: \( a \rightarrow \Delta a \rightarrow a \)
- \((\triangleright) \) :: \( a \rightarrow a \rightarrow \Delta a \)
- \(0 \) :: \( \Delta a \)

The unit type has a trivial instance, since there can be no changes; a change for pairs is a pair of changes; and a change for functions is a function to changes. Atomic types can be handled in various ways, including simply a \texttt{Maybe} giving a new value if changed.

A change morphism says how to map changes to changes:

\[
\text{newtype} \text{DelX} a b = \text{DelX} (\Delta a \rightarrow \Delta b)
\]
It’s easy to then give Category and Cartesian instances for DelX:

```haskell
instance Category DelX where
  type Ok DelX = HasDelta
  id = DelX id
  DelX g ∘ Del f = Del (g ∘ f)

instance Cartesian DelX where
  exl = DelX exl
  expr = DelX expr
  DelX f △ Del g = Del (f △ g)
```

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

```haskell
type Inc = GD DelX
```

### 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 \( I \), \( ∀x ∈ I. f(x) ∈ \hat{f}(I) \). 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:

```haskell
data IFun a b = IFun (Interval a → Interval b)

instance Closed IFun where
  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:

```haskell
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-kindled 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 operations elegantly defined by 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:

\[
\text{infixr 2 } ∨
\]

\[
\text{class } \text{Category } k \Rightarrow \text{Cocartesian } k \text{ where}
\]

\[
\text{inr} :: \text{Ok}_k \ a \ b \Rightarrow a' \ k' \ (a + b)
\]

\[
\text{(∨)} :: \text{Ok}_k \ a \ c \ d
\]

\[
\Rightarrow (c' \ k') \ a \rightarrow (d' \ k') \ a \rightarrow ((c + d) \ 'k \ a)
\]

The universal property is also dual to that of Cartesian:

\[
∀h. \ h ≡ f ∨ g ⇐⇒ h \circ \text{inl} ≡ f \land h \circ \text{inr} ≡ g
\]

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:

\[
\text{class } \text{(Cartesian } k, \text{Cocartesian } k) \Rightarrow \text{Distrib } k \text{ where}
\]

\[
\text{distl} :: \text{Ok}_k \ a \ u \ v
\]

\[
\Rightarrow (a \times (u + v))' k' (a \times u + a \times v)
\]

\[
\text{distr} :: \text{Ok}_k \ a \ u \ v
\]

\[
\Rightarrow (a \times u + a \times v)' k' (a \times (u + v))
\]

We can define either \( \text{distl} \) or \( \text{distr} \) 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:

\[
(λx → \text{case } \text{ scrut of } \{ \text{Left } u → U; \text{Right } v → V \}) ≡ (λx → (U ∨ V) \text{ scrut})
\]

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

\[
(λx → U ∨ V) ≡ \text{curry } ((\text{uncurry } (λx → U) ∨ \text{uncurry } (λx → V)) \circ \text{distl})
\]

The proof is left as an exercise.
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:

```haskell
class HasRep a where
  -- Law: abst \circ repr \equiv id
  type Rep a
  repr :: a \rightarrow Rep a
  abst :: Rep a \rightarrow a
```

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

```haskell
instance HasRep (Pair a) where
  type Rep (Pair a) = a \times 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 \ldots e_n`, rewrite it to `abst (inline repr (Con e_1 \ldots e_n))`, where `inline e` tells GHC’s simplifier to inline the expression `e`.

GHC’s usual simplifications then eliminate the constructor, leaving only `abst` behind. For example,

```
P 3 4
≡ {\{ abst \circ repr \equiv id \}}
abst (inline repr (P 3 4))
≡ {\{ inline repr \}}
abst (\lambda p \rightarrow case p of P a a' \rightarrow (a, a')) (P 3 4)
≡ {\{ \beta\text{-reduce} \}}
abst (let p = P 3 4 in case p of P a a' \rightarrow (a, a'))
≡ {\{ let\text{-substitute} \}}
abst (case P 3 4 of P a a' \rightarrow (a, a'))
≡ {\{ GHC’s case-of-known-constructor transformation \}}
abst (3, 4)
```

Dually, consider a case expression `case scrut of \{ p_1 \rightarrow rhs_1; \ldots; 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,

```
  case p of P x y \rightarrow x + y
≡ {\{ abst \circ repr \equiv id \}}
```

---

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 \eta\text{-expanded until saturated.}
case inline abst (repr p) of P x y → x + y
≡ {- inline abst -}
case (λq → case q of (a, a') → P a a') (repr p) of P x y → x + y
≡ {- β-reduce and let-substitute -}
case (case repr p of (a, a') → P x y → x + y
≡ {- GHC’s case-of-case transformation -}
case repr p of (a, a') → let { x = a; y = a'} in x + y
≡ {- let-substitute -}
case repr p of (a, a') → a + a’

These two transformations occur only in the context “ccc (λx → ...)”. 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

  reprC :: a 'k' Rep a
  abstC :: Rep a 'k' a

instance HasRep a ⇒ RepCat (→) a where

  reprC = repr
  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.

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 (→) 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)

Elliott [2017] showed many examples with non-standard types compiled to Graph.

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 Intₙ
Numerical operations on \( \text{Int} \) are then defined to unwrap (unbox) arguments, apply unboxed operations on the contents, and rewrap (rebox) the unboxed results:

\[
\text{instance } \text{Num } \text{Int} \text{ where } \\
\text{fromInteger } i = I_\text{b} \text{ (integerToInt } i) \\
\text{negate } (I_\text{b} x) = I_\text{b} \text{ (negateInt } x) \\
I_\text{b} x + I_\text{b} y = I_\text{b} (x +_\text{b} y) \\
I_\text{b} x - I_\text{b} y = I_\text{b} (x -_\text{b} y) \\
I_\text{b} x * I_\text{b} y = I_\text{b} (x *_\text{b} y)
\]

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 \( a, b :: \text{Int} \) the expression “\( a + 3 \times b \)” optimizes to the following:

\[
\text{case } a \text{ of } I_\text{b} x \rightarrow \text{case } b \text{ of } I_\text{b} y \rightarrow I_\text{b} (x +_\text{b} 3 \times_\text{b} y)
\]

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 (\(\to\)) category), it complicates conversion to categorical form and hence compiling to other categories. Recall the signatures involved in the generalized \text{Num} class:

\[
\text{class } \text{NumCat } k a \text{ where } \\
\text{addC}, \text{subC}, \text{mulC} :: (a \times a) \to k a \\
\]

While these methods are polymorphic over number type/object, the polymorphism is implicitly restricted to kind \( * \), i.e., \text{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 \( \text{I}_{\text{b}} \), such as \( \text{I}_{\text{b}} (x +_\text{b} 3 \times_\text{b} y) \) in the optimized example above. Replace those outer constructors with a synonym defined as follows:

\[
\text{boxI} :: \text{Int}_{\text{b}} \to \text{Int} \\
\text{boxI} = I_\text{b} \\
\text{{-# INLINE [0] boxI #-}}
\]

The \text{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

\[
\text{case } a \text{ of } I_\text{b} x \rightarrow \text{case } b \text{ of } I_\text{b} y \rightarrow \text{boxI} (x +_\text{b} 3 \times_\text{b} y)
\]

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

\[
\forall u. \text{ boxI} (\text{negateInt}_{\text{b}} u) = \text{negateC} (\text{boxI} u) \\
\forall u v. \text{ boxI} (u +_\text{b} v) = \text{addC} (\text{boxI} u, \text{boxI} v) \\
\forall u v. \text{ boxI} (u -_\text{b} v) = \text{subC} (\text{boxI} u, \text{boxI} v) \\
\forall u v. \text{ boxI} (u *_\text{b} v) = \text{mulC} (\text{boxI} u, \text{boxI} v)
\]

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.
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

\[
\text{case } a \text{ of } \_ \rightarrow \text{case } b \text{ of } \_ \rightarrow \text{addC (boxI } x, \text{mulC (boxI } 3, \text{boxI } y))
\]

One more transformation eliminates the unboxing case scrutinees: transform an expression like “case a of \_ \rightarrow ...boxI x...” to “let \(x' = a\) in ...\(x'\)...”. 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:

\[
\text{addC (a, mulC (boxI } 3, \text{b))}
\]

### 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 \((\to)\) 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:

\[
\forall f. \quad \text{uncurry (curry } f) = f
\]
\[
\forall g. \quad \text{curry (uncurry } g) = g
\]
\[
\forall g f. \quad \text{apply } \circ (\text{curry (g } \circ \text{exr) } \triangle f) = g \circ f
\]

### 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:

- 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, 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 $\lambda$-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.

C$\_\lambda$aSH 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, C$\_\lambda$aSH compiles a somewhat restricted form of Haskell rather than being a Haskell library implemented as a deep or shallow embedding. Unlike the present paper, C$\_\lambda$aSH only compiles to hardware.

Cai et al. [2014] describes how to convert a typed $\lambda$-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:

- 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 $\ast$. The only reason for this restriction is the fixed versions of product, unit, etc. Once those types are generalized,
they can easily become poly-kindled [Hinze 2004; 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 \( f \, @ \, v_1 \ldots \, @ \, v_n = \text{rhs} \) (where \( v_1, \ldots, v_n \) are type variables), the compiler plugin generated a rewrite rule \( \text{ccc} \, (\, f \, @ \, v_1 \ldots \, @ \, v_n \, ) = \, \text{ccc} \, \text{rhs} \). The right-hand side of this rule simplifies to some term \( \text{rhs}' \), often containing residual \( \text{ccc} \) applications due to polymorphism. Later, when a type instance \( \text{ccc} \, (\, f \, @ \, \tau_1 \ldots \, @ \, \tau_n \, ) \) 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 \( \text{rhs}' \), 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 \( \text{Graph} \) 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 \( \text{Bool} \) and \( \text{Int} \) 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. While the compiler plugin implementation is non-trivial, working with internal
compiler representations, it is done once per host compiler (e.g., GHC) rather than per DSEL.

The technique described in this paper has been illustrated in terms of Haskell and its compiler
GHC, but it could be applied to other languages and compilers as well. A few properties of GHC
have been particularly helpful:

- The whole source language reduces to a small core typed λ-calculus [Sulzmann et al. 2007].
- Optimization mainly comprises transformations on this core language [Peyton Jones 1996].
- The set of such transformations is extensible by a compiler plugin [GHC Team 2016]. (Given
  the generality of the needed additional transformations, however, one might build them into
  an existing compiler instead of writing a plugin. Doing so would require coordination with
  and support from the compiler’s implementors, maintainers, and community.)

As for language features, type classes are very useful as a way to package the required inter-
faces (Category, Cartesian, NumCat, etc) as well as their specific definitions for alternative
interpretations/categories. Especially important here is that these interpretations are defined in
familiar-looking source code entirely outside of the compiler and plugin, making it easy to add new
interpretations, such as those of Section 7, without any knowledge of compiler internals (including
the existence or particulars of GHC Core).

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 → s for some type A. For
instance, the vector space \( \mathbb{R}^2 \) over \( \mathbb{R} \) is represented as a Pair \( \mathbb{R} \), where Pair is a uniform pair functor,
which is isomorphic to \( \text{Bool} \to \mathbb{R} \). This representation enables simple and general definitions of
“vector” operations, e.g.,

\[
\text{scaleV} :: (\text{Functor } f, \text{Num } s) \Rightarrow s \to f s \to f s \quad \text{-- Scale a vector}
\]

\[
s \cdot \text{scaleV} \; v = (s \ast) \triangleleft v
\]

\[
\text{addV} :: (\text{Zip } f, \text{Num } s) \Rightarrow f s \to f s \to f s \quad \text{-- Add vectors}
\]

\[
\text{addV} = \text{zipWith} (+)
\]

\[
\text{dotV} :: (\text{Zip } f, \text{Foldable } f, \text{Num } s) \Rightarrow f s \to f s \to s \quad \text{-- Dot product}
\]

\[
x \cdot \text{dotV} \; y = \text{sum} (\text{zipWith} (\ast) \; x \; y)
\]

\[
\text{zeroV} :: (\text{Pointed } f, \text{Num } a) \Rightarrow f a \quad \text{-- Zero vector}
\]

\[
\text{zeroV} = \text{point} 0
\]
The \textit{Pointed} class has one method $point : s \to f \ s$ that fills a structure with a given value. The \textit{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 \textit{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):

\begin{verbatim}
infixr 1 ⊸
type (a ⊸ b) s = b (a s)
\end{verbatim}

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

\begin{verbatim}
lapplyL :: (Zip a, Foldable a, Zip b, Num s) ⇒ (a ⊸ b) s → a s → b s
lapplyL a = (‘dotV‘ a) ⊸ as
\end{verbatim}

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

\begin{verbatim}
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
\end{verbatim}

The \textit{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 \textit{Category} instance in Section 7.4, the following three are used in the \textit{Cartesian} instance:

\begin{verbatim}
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 = (×)
\end{verbatim}

Finally, the function $\texttt{linearL}$ that converts from a function (presumably linear) to the functor representation of linear maps:

\begin{verbatim}
class OkLF f ⇒ HasL f where
    linearL :: ∀s g.(Num s, OkLF g) ⇒ (f s → g s) → (f ⊸ g) s
    -- Law: linearL ∘ lapplyL ≡ id
instance HasL U₁ where linearL h = const U₁ ⊸ h U₁
instance HasL Par₁ where linearL h = Par₁ ⊸ h (Par₁ 1)
\end{verbatim}

\begin{verbatim}
instance (HasL f, HasL g) ⇒ HasL (f ✕ g) where
    linearL h = zipWith (×) (linearL (h ∘ (× zeroV))) (linearL (h ∘ (× zeroV))
\end{verbatim}

The constraint used in defining $\texttt{Ok}$ (\texttt{− ⊸}) conjoins required class constraints:

\begin{verbatim}
type OkLF a = (Foldable a, Pointed a, Zip a, Diagonal a)
\end{verbatim}
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 am 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 [Farmer et al. 2012; Sculthorpe et al. 2013b], on which earlier implementations of the Haskell-to-hardware plugin were built. 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

Haskell library.

Geraint Jones and Mary Sheeran. Circuit design in Ruby. Formal methods for VLSI design, 1, 1990.
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, 2005.