An Ode to Arrows

Hai Liu   Paul Hudak

Department of Computer Science
Yale University
New Haven, CT 06520, U.S.A.
{hai.liu,paul.hudak}@yale.edu

Abstract. We study a number of embedded DSLs for *autonomous ordinary differential equations* (autonomous ODEs) in Haskell. A naive implementation based on the *lazy tower of derivatives* is straightforward but has serious time and space leaks due to the loss of sharing when handling cyclic and infinite data structures. In seeking a solution to fix this problem, we explore a number of DSLs ranging from shallow to deep embeddings, and middle-grounds in between. We advocate a solution based on *arrows*, an abstract notion of computation that offers both a succinct representation and an effective implementation. *Arrows* are ubiquitous in their combinator style that happens to capture both sharing and recursion elegantly. We further relate our arrow-based DSL to a more constrained form of arrows called *causal commutative arrows*, the normalization of which leads to a staged compilation technique improving ODE performance by orders of magnitude.

1 Introduction

Consider the following stream representation of the "lazy tower of derivatives" [9] in Haskell:

$$\textbf{data}\ D\ a = D\ \{ \mathit{val} :: a, \mathit{der} :: D\ a \}\ \textbf{deriving}\ (\mathit{Eq}, \mathit{Show})$$

Mathematically it represents an infinite sequence of derivatives $f(t_0)$, $f'(t_0)$, $f''(t_0)$, $\ldots$, $f^{(n)}(t_0)$, $\ldots$ for a function f that is continuously differentiable at some value t_0. This representation has been used frequently in a technique called *Functional Automatic Differentiation* The usual trick in Haskell is to make $D\ a$ an instance of the *Num* and *Fractional* type classes, and overload the mathematical operators to simultaneously work on all values in the tower of derivatives:

$$\begin{array}{ll}
\multicolumn{2}{l}{\textbf{instance}\ \mathit{Num}\ a \Rightarrow \mathit{Num}\ (D\ a)\ \textbf{where}} \\
\quad D\ x\ x' + D\ y\ y' & = D\ (x + y)\ (x' + y') \\
\quad u@(D\ x\ x') * v@(D\ y\ y') & = D\ (x * y)\ (x' * v + u * y') \\
\quad \mathit{negate}\ (D\ x\ x') & = D\ (-x)\ (-x') \\
\quad \ldots &
\end{array}$$

Sine wave	$y'' = -y$	$y = \mathit{init}\ y_0\ y'$ $y' = \mathit{init}\ y_1\ (-y)$
Damped oscillator	$y'' = -cy' - y$	$y = \mathit{init}\ y_0\ y'$ $y' = \mathit{init}\ y_1\ (-c * y' - y)$
Lorenz attractor	$x' = \sigma(y - x)$ $y' = x(\rho - z) - y$ $z' = xy - \beta z$	$x = \mathit{init}\ x_0\ (\sigma * (y - x))$ $y = \mathit{init}\ y_0\ (x * (\rho - z) - y)$ $z = \mathit{init}\ z_0\ (x * y - \beta * z)$

Fig. 1. A few ODE examples

1.1 Autonomous ODEs and the Tower of Derivatives

Our first contribution is a simple but novel use of the "lazy tower of derivatives" to implement a domain specific language (DSL) for *autonomous ordinary differential equations* (autonomous ODEs). Mathematically, an equation of the form:

$$f^{(n)} = F(t, f, f', \ldots, f^{(n-1)})$$

is called an ordinary differential equation of order n for an unknown function $f(t)$, with its n^{th} derivative described by $f^{(n)}$, where the types for f and t are $\mathbb{R} \rightarrow \mathbb{R}$ and $\mathbb{R}$ respectively. A differential equation not depending on t is called *autonomous*. An *initial value problem* of a first order autonomous ODE is of the form:

$$f' = F(f) \quad s.t. \quad f(t_0) = f_0$$

where the given pair $(t_0, f_0) \in \mathbb{R} \times \mathbb{R}$ is called the *initial condition*. The solution to a first-order ODE can be stated as:

$$f(t) = \int f'(t)dt + C$$

where C is the *constant of integration*, which is chosen to satisfy the initial condition $f(t_0) = f_0$.

In Haskell we represent the above integral operation as *init* that takes an initial value f_0:

```
init :: a → D a → D a
init = D
```

As an example, consider the simple ODE $f' = f$, whose solution is the well known exponential function, and can be defined in terms of *init*:

```
e = init 1 e
```

which is a valid Haskell definition that evaluates to a concrete value, namely, a recursively defined tower of derivatives, starting from an initial value of 1, with its derivative equal to itself.

In general, by harnessing the expressive power of recursive data types and overloaded arithmetic operators, we can directly represent autonomous ODEs

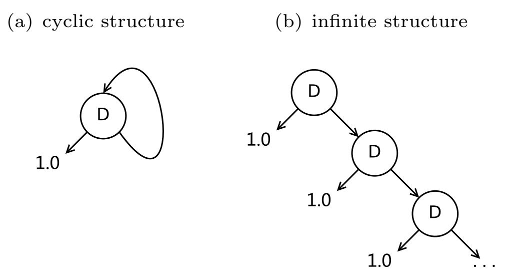

Fig. 2. Two structural diagrams for e

as a set of Haskell definitions. We give a few more examples in Figure 1. Note that in the sine wave and damped oscillator examples, we translate higher-order ODEs into a system of first-order equations.

The solution to the initial value problem of an ODE can often be approximated by numerical integration. Here is a program that integrates a tower of derivatives at t_0 to its next step value at $t_0 + h$ using the Euler method:

```
euler     :: Num a ⇒ a → D a → D a
euler h f = D (val f + h ∗ val (der f)) (euler h (der f))
```

The function *euler* lazily traverses and updates every value in the tower of derivatives by their next step values. By repeatedly applying *euler*, we can sample the approximate solution to an ODE:

```
sample   :: Num a ⇒ a → D a → [a]
sample h = map val . iterate (euler h)
```

For instance, evaluating *sample* 0.001 e generates an infinite sequence of the exponential function $\exp(t)$ sampled at a 0.001 interval starting from $t = 0$:

```
[1.0, 1.001, 1.002001, 1.003003001, 1.004006004001, ...
```

1.2 Time and Space Leaks

Thus far, we have designed a DSL embedded in Haskell for autonomous ODEs. However, our DSL, despite its elegant implementation, has but one problem: *the numerical solver has serious time and space leaks.* For instance, unfolding the sequence *sample* 0.001 e in GHCi exhibits a quadratic time behavior instead of linear. Evaluating more complex definitions than e can exhibit even worse leaks.

The problem is that data sharing is lost when we update a recursive structure [10]. In a lazy and pure functional setting, cyclic and infinite data structures are indistinguishable when they semantically denote the same value, as illustrated in Figure 2. Usually an implementation of a lazy language allows one to "tie the knot" using recursive definitions such as $e = init\ 1\ e$, which would create an internal data structure as pictured in Figure 2(a). This kind of knot tying, however, is very limited, and even the simplest traversal like the one below loses sharing:

$$id\ (D\ v\ d) = D\ v\ (id\ d)$$

When evaluating $id\ e$, a lazy (call-by-need) strategy fails to recognize that in the unfolding of $id\ e = id\ (D\ 1\ e) = D\ 1\ (id\ e)$, the last and first occurrences of $id\ e$ could share the same value, and therefore produces something like in Figure 2(b). Repeatedly evaluating an update function such as *euler* on a recursively defined value of type $D\ a$ will force unfolding the structure indefinitely, and hence create leaks both in space and time.

In the remainder of this paper we embark on a journey seeking the best way to implement our DSL for ODEs with varying degrees of embedding. Specifically, our paper makes the following contributions:

1. We study the problem of handling cyclic and infinite structures by analyzing different DSL representations and implementations, from shallow to deep embeddings, and mid-grounds in between.
2. We present an arrow-based DSL that captures sharing implicitly but without the usual deficiency of having to observe and compare equivalences using tags or references. Additionally the use of *arrow notation* [15] enables succinct syntax for ODEs.
3. We illustrate that sharing and recursion in an object language can be better captured by arrows than higher-order abstract syntax (HOAS), even though both are mixing shallow and deep embeddings.
4. We make use of the arrow properties, and specifically the normal form of *causal commutative arrows* (CCA) [11], to compile our DSL and eliminate all overhead introduced by the abstraction layer.

2 Sharing of Computation

2.1 A Tagged Solution

To distinguish cyclic from infinite data structures, we can make the sharing of sub-structures explicit by labeling them with unique tags [14]. The traversal of a tagged structure must keep track of all visited tags and skip those that are already traversed in order to avoid endless loops.

It must be noted, however, that not all infinite data structures can be made cyclic. This can be demonstrated by the multiplication of two towers of derivatives $x, x', \ldots, x^{(m-1)}, \ldots$ and $y, y', \ldots, y^{(n-1)}, \ldots$, which produces the following sequence:

$$\begin{array}{l} xy \\ x'y + xy' \\ x''y + x'y' + x'y' + xy'' \\ \ldots \end{array}$$

Even if both sequences of x and y are cyclic ($x^{(i)} = x^{(i \ mod \ m)}$, $y^{(j)} = y^{(j \ mod \ n)}$, for all $i \geq m, j \geq n$), the resulting sequence does not necessarily have a repeating pattern that loops over from the beginning, or any part in the middle. Therefore merely adding tags to the tower of derivatives is not enough; we need to represent mathematical operations symbolically so that they become part of the data structure and hence subject to traversal as well. For instance:

```
data C a    = CI a    (T a)           -- init operator
            | C1 Op (T a)             -- unary arithmetic
            | C2 Op (T a) (T a)       -- binary arithmetic
type T a    = Tag (C a)
data Tag a = Tag Int a
type Op     = String
```

This is a simple DSL that supports initialization (*CI*) in addition to both unary (*C1*) and binary (*C2*) operations. Since every node in a (*T a*) structure is tagged, we can easily detect sharing or cycles by comparing tags. There are different ways to generate unique tags; we follow Bjesse et al. [2] and use a state monad: [1]

```
type M a = State Int (T a)      -- monad that returns T a
newtag     :: State Int Int      -- to get fresh new tag
newtag     = modify (+1) >> get
tag        :: C a → M a          -- tag a node with new tag
tag x      = newtag >>= λi → return (Tag i x)
initT      :: a → T a → M a      -- init with a new tag
initT v d  = tag (CI v d)
```

Since our DSL now represents all operations as part of its data structure, we no longer need the chain rule to evaluate multiplication, and instead we just represent it symbolically. Such a technique is often called *deep embedding* in contrast to our first DSL, which is a *shallow embedding* since all its operators are ordinary Haskell functions. We leave the rest of the implementation to our readers.

With the same exponential example now defined as $e = mfix\ (initT\ 1)$, [2] repeatedly sample its value in GHCi now exhibits linear time behavior, and runs in constant space as one would have expected. By moving from shallow to deep embedding, and with the help of tags, we are now able to recover sharing in the interpretation of our tagged DSL.

2.2 Higher Order Abstract Syntax

Although the tagged solution successfully avoids space leaks, it is cumbersome due to the overhead of generating and maintaining unique tags. One way to avoid

[1] The *State* type and functions like *modify* and *get* are from the standard Haskell module *Control.Monad.State*.

[2] Function *mfix* computes the fixed point of a monad, and is of type $MonadFix\ m \Rightarrow (a \rightarrow m\ a) \rightarrow m\ a$

dealing with tags is to mimic *Let*-expressions for sharing, and *Letrec* for recursion. However, *Let*-expressions in the object language require variable bindings and their interpretations. Indeed, variables are just lexically scoped tags, and they are remembered in an environment instead of a state monad.

An alternative solution that avoids variable bindings in the object language is to use higher-order abstract syntax (HOAS). For example, we may modify our DSL to include both *Let* and *Letrec* as follows:

```
data H a = HI     a      (H a)          -- init operator
         | H1     Op     (H a)          -- unary operator
         | H2     Op     (H a)   (H a)  -- binary operator
         | Let    (H a → H a) (H a)
         | LetRec (H a → H a)
         | Var    Int                   -- for internal use only
```

Where *Let* f x introduces the sharing of x in the result of $f\ x$, and *LetRec* f introduces an explicit cycle in computing the fixed point of f. When traversing *Let* and *LetRec*, however, we have to remember shared values for later lookups in an environment. For this reason we need to use *Var* i to represents an index i in such an environment. We leave the actual implementation of this DSL to our readers.

Now we can define the same exponential ODE as *LetRec* (*init* 1) where *init* = *HI*. But the real trouble comes when we want to update it in the *euler* function. Here is a sample code snippet that updates a *Let* structure:

```
update env (Let f x) =
  let x'   = update env x
      i    = length env
      f' y = update ((i, (y, valH env x)) : env) (f (Var i))
  in  Let f' x'
```

The function *update* remembers shared values in an environment variable *env* during a traversal. To update a value of *Let* f x is to create a new function f' out of f in some way, and return *Let* f' x'. In computing f' it must reference the environment to get the shared value of x using *valH env* x. Therefore f' is really a new closure. Since our host language Haskell is not able to introspect or evaluate under lambdas, repeatedly updating HOAS structures in this way will result in building larger and larger closures, and hence creating a new kind of space leak. A possible remedy to this situation is *memoization* [12]. For example, we can have a pair of conversion functions between the HOAS language and the tagged language:

```
toT   :: H a → T a
fromT :: T a → H a
```

Computation over $H\ a$ can then be expressed in terms of computations over $T\ a$. As a result of *toT*, the intermediate tagged structure is of fixed size (relative to the input), and hence *fromT* will create a HOAS structure also of fixed size.

Unfortunately, this approach introduces considerably more runtime overhead and begins to feel just as cumbersome as tagging. Therefore we consider HOAS inadequate as a technique for object languages that require careful sharing.

For our next and final DSL, we represent then computation between derivatives in an ODE as *arrows*. But before doing so, we first give an introduction to arrows. Readers familiar with this topic may skip to Section 4.

3 An Introduction to Arrows

Arrows [8] are a generalization of monads that relax the stringent linearity imposed by monads, while retaining a disciplined style of composition. Arrows have enjoyed a wide range of applications, often as an embedded DSL, including signal processing [13], graphical user interface [4], and so on.

3.1 Conventional Arrows

Like monads, arrows capture a certain class of abstract computations, and offer a way to structure programs. This is achieved through the *Arrow* type class:

$$\begin{array}{l}
\textbf{class}\ \mathit{Arrow}\ a\ \textbf{where} \\
\quad \mathit{arr}\ \ :: (b \to c) \to a\ b\ c \\
\quad (\ggg) :: a\ b\ c \to a\ c\ d \to a\ b\ d \\
\quad \mathit{first}\ :: a\ b\ c \to a\ (b, d)\ (c, d)
\end{array}$$

The combinator *arr* lifts a function from b to c to a “pure” arrow computation from b to c, namely $a\ b\ c$ where a is the arrow type. The output of a pure arrow entirely depends on the input (it is analogous to *return* in the *Monad* class). $\ggg$ composes two arrow computations by connecting the output of the first to the input of the second (and is analogous to bind $((\gg=))$ in the *Monad* class). But in addition to composing arrows linearly, it is desirable to compose them in parallel – i.e. to allow “branching” and “merging” of inputs and outputs. There are several ways to do this, but by simply defining the *first* combinator in the *Arrow* class, all other combinators can be defined. The combinator *first* applies an arrow to the first part of the input, and the result becomes the first part of the output. The second part of the input is fed directly to the second part of the output.

Other combinators can be defined using these three primitives. For example, the dual of *first* can be defined as:

$$\begin{array}{l}
\mathit{second} :: \mathit{Arrow}\ a \Rightarrow a\ b\ c \to a\ (d, b)\ (d, c) \\
\mathit{second}\ f = \mathit{arr}\ \mathit{swap} \ggg \mathit{first}\ f \ggg \mathit{arr}\ \mathit{swap} \\
\quad \textbf{where}\ \mathit{swap}\ (a, b) = (b, a)
\end{array}$$

Parallel composition can be defined as a sequence of *first* and *second*:

$$\begin{array}{l}
(\star\!\star\!\star) :: \mathit{Arrow}\ a \Rightarrow a\ b\ c \to a\ b'\ c' \to a\ (b, b')\ (c, c') \\
f \star\!\star\!\star\ g = \mathit{first}\ f \ggg \mathit{second}\ g
\end{array}$$

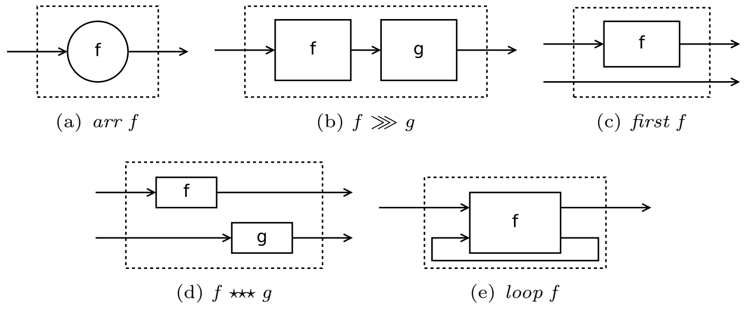

Fig. 3. Commonly used arrow combinators

To model recursion, we can introduces a *loop* combinator [15], which is captured in the *ArrowLoop* class.

$$
\begin{array}{l}
\textbf{class } \mathit{Arrow}\ a \Rightarrow \mathit{ArrowLoop}\ a \textbf{ where} \\
\quad \mathit{loop} :: a\ (b, d)\ (c, d) \rightarrow a\ b\ c
\end{array}
$$

We find that arrows are best viewed pictorially. Figure 3 shows some of the basic combinators in this manner, including *loop*. A mere implementation of the arrow combinators, of course, does not make it an arrow – the implementation must additionally satisfy a set of *Arrow and ArrowLoop laws*, which are omitted here for the lack of space. See [8, 15] for further details.

3.2 Arrow Notation

Arrow expressions we have seen so far maintain a point-free style that requires explicit "plumbing" using arrow combinators, and may be obscure and inconvenient in some cases. Paterson [15] devises a set of *arrow notation* that help users to express arrows in a "point-ful" style with improved presentation. Programs written in such special syntax can be automatically translated by a pre-processor back to the combinator form. GHC in fact has built-in support for arrow notations.

For space reasons we omit translation rules of arrow notation, and instead we briefly explain through the example of the parallel composition $\star\!\star\!\star$ as follows:

$$
\begin{array}{ll}
(\star\!\star\!\star) & :: \mathit{Arrow}\ a \Rightarrow a\ b\ c \rightarrow a\ b'\ c' \rightarrow a\ (b, b')\ (c, c') \\
f \star\!\star\!\star\ g & = \textbf{proc}\ (x, y) \rightarrow \textbf{do} \\
\multicolumn{2}{l}{\quad x' \leftarrow f \prec x} \\
\multicolumn{2}{l}{\quad y' \leftarrow g \prec y} \\
\multicolumn{2}{l}{\quad \mathit{returnA} \prec (x', y')} \\
\multicolumn{2}{l}{\mathit{returnA} :: \mathit{Arrow}\ a \Rightarrow a\ b\ b} \\
\multicolumn{2}{l}{\mathit{returnA} = \mathit{arr}\ (\lambda x \rightarrow x)}
\end{array}
$$

The **proc** keyword starts an arrow expression whose input is a pair (x, y), and whose output is the output of the last command in the **do**-block. The **do**-block

Sine wave	$y'' = -y$	$\textbf{proc}\ () \rightarrow \textbf{do}$ $\quad \textbf{rec}\ y \leftarrow \mathit{init}\ y_0 \prec y'$ $\qquad y' \leftarrow \mathit{init}\ y_1 \prec -y$ $\quad \mathit{returnA} \prec y$
Damped oscillator	$y'' = -cy' - y$	$\textbf{proc}\ () \rightarrow \textbf{do}$ $\quad \textbf{rec}\ y \leftarrow \mathit{init}\ y_0 \prec y'$ $\qquad y' \leftarrow \mathit{init}\ y_1 \prec -c * y' - y$ $\quad \mathit{returnA} \prec y$
Lorenz attractor	$x' = \sigma(y - x)$ $y' = x(\rho - z) - y$ $z' = xy - \beta z$	$\textbf{proc}\ () \rightarrow \textbf{do}$ $\quad \textbf{rec}\ x \leftarrow \mathit{init}\ x_0 \prec \sigma * (y - x)$ $\qquad y \leftarrow \mathit{init}\ y_0 \prec x * (\rho - z) - y$ $\qquad z \leftarrow \mathit{init}\ z_0 \prec x * y - \beta * z$ $\quad \mathit{returnA} \prec (x, y, z)$

Fig. 4. ODE examples in arrow notation

allows one to use variable bindings as "points" to interconnect arrows, e.g., $x' \leftarrow f \prec x$ passes a value x through an arrow f and names the result x'. So the **proc** expression is really just another way to express arrow compositions by naming the "points", in contrast to the point-free style.

4 ODE and Arrows

We begin with an abstract view of ODE programs without committing to a particular arrow implementation. Here is the exponential ODE example written in arrow notation:

```
e = proc () → do
  rec e ← init 1 ≺ e
  returnA ≺ e
```

In the above program, the **rec** keyword indicates a recursive definition, We give more examples in Figure 4 by re-writing in arrow notation the same ODEs given in Figure 1.

In the actual implementation, we simply lift all arithmetic operations to pure arrows, and the only domain specific operator needed is an *init* arrow. Following our previous two DSL designs, we have to traverse the internal structure of our DSL and update all initial values. Hence a natural choice is to implement our arrow to reflect this kind of traversal:

```
newtype ODE s a b = ODE (Updater s → a → (b, ODE s a b))
type    Updater s  = s → s → s
```

The *ODE* type is parameterized by the type of initial value s, and implemented as a function that takes an *Updater* and an input value of type a, and returns a pair: output value of type b, and an updated ODE. The only place we actually apply the *Updater* is in the *init* combinator, where both the initial value and the current input are given to the *Updater* to produce an updated initial value:

```
init   :: s → ODE s s s
init i = ODE h
  where h f x = (i, init (f i x))
```

All other arrow combinators simply pass the *Updater* around to complete a full traversal. Then we can perform numerical integrations by passing the *euler* function as the *Updater*, and implement the *sample* function in a similar way as we have seen before:

```
instance Arrow (ODE s) where
  arr f               = ODE h where h u x      = (f x, arr f)
  ODE f ⋙ ODE g     = ODE h where h u x      = let (y, f′) = f u x
                                                     (z, g′) = g u y
                                                 in  (z, f′ ⋙ g′)
  first (ODE f)       = ODE h where h u (x, z) = let (y, f′) = f u x
                                                 in  ((y, z), first f′)
instance ArrowLoop (ODE s) where
  loop (ODE f)        = ODE h where h u x      = let ((y, z), f′) = f u (x, z)
                                                 in  (y, loop f′)
euler       :: Num s ⇒ s → Updater s
euler h i x = i + h ∗ x
sample      :: Num s ⇒ s → ODE s () c → [c]
sample h (ODE f) = y : sample h f′
  where (y, f′) = f (euler h) ()
```

This approach is not only elegant, it is also efficient – there are no space leaks. For example, unfolding *sample* 0.001 *e* in GHCi executes correctly and exhibits a linear time behavior. This is because

1. The representation of an ODE is composed from a fixed number of arrows with no cycles, and thus the traversal will always terminate.
2. Although the arrow itself is implemented as a higher-order function, unlike the HOAS implementation, it makes no references to environment values, and hence it is not a closure.
3. The traversal of all arrows returns new arrows of the same size, which can be proved by a structural induction as follows:
   (a) The traversal of a pure arrow always returns a pure arrow of the same size.
   (b) The traversal of all arrow compositions (⋙, *first*, and *loop*) always returns a composition of the same structure, and of the same size.
   (c) The update of initial values is only within the *init* arrow, which also returns a new arrow of the same size.

Of course the above is only an informal proof; a formal proof would depend on a more precise definition of size, and the lazy (call-by-need) semantics of the host

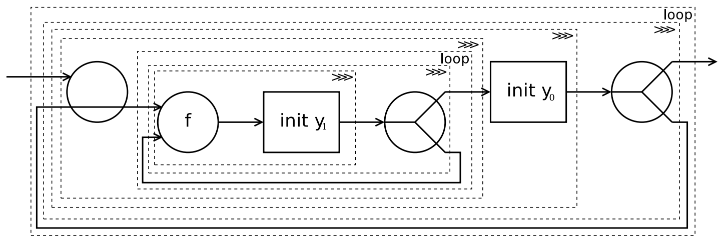

Fig. 5. Arrow diagram of damped oscillator

language. We omit such proofs here. It must be noted, however, that much of the above reasoning has little to do with the actual implementation of the arrow and its combinators. In other words, *arrows capture sharing by design.*

This intuition becomes more evident when we look at arrow programs written using combinators. As a slightly more complex example, we translate the program for a damped oscillator given in Figure 4 to combinators below:

$$\begin{aligned}&\mathit{loop}\ (\mathit{arr}\ \mathit{snd} \ggg \mathit{loop}\ (\mathit{arr}\ f \ggg \mathit{init}\ y_1 \ggg \mathit{arr}\ \mathit{dup}) \ggg \mathit{init}\ y_0 \ggg \mathit{arr}\ \mathit{dup})\\ &\quad \textbf{where}\ \mathit{dup}\ x \quad = (x, x)\\ &\qquad\qquad f \quad (y, y') = -c * y' - y\end{aligned}$$

It is obvious that the above program consists of a fixed number of arrows that are easy to traverse or manipulate. The same program is presented pictorially in Figure 5 where the loops represent the values of y (outer) and y' (inner) being fed back to the inputs. Their values are shared at all the "points". For instance, the function *dup* only evaluates its argument once.

Both HOAS and arrow-based DSLs can be viewed as middle grounds between shallow and deep embeddings. We advocate the use of arrows because, Unlike HOAS, lambdas in the object language are represented as compositions of arrow combinators, which lends to easy program manipulation. Also, We no longer have to deal with variable bindings, environments or open terms since all arrows translate to combinators that are always closed, and do not require memoization.

5 ODE and CCA

The use of the *init* arrow combinator is interesting – it introduces an internal state that is subject to both intentional computation (for being an arrow) and extensional examination (for being part of a traversal). If we ignore the monomorphism restriction of the *ODE* arrow for a moment, we can make a further abstraction by defining a new type class:

$$\begin{aligned}&\textbf{class}\ \mathit{Arrow}\ a \Rightarrow \mathit{ArrowInit}\ a\ \textbf{where}\\ &\quad \mathit{init} :: b \rightarrow a\ b\ b\end{aligned}$$

The *ArrowInit* class actually represents a more constrained arrow called *Causal Commutative Arrow* (CCA) [11] that builds on top of a simply typed lambda calculus (with a few extensions), and must satisfy two additional laws besides the arrow and arrow loop laws:

commutativity $\mathit{first}\ f \ggg \mathit{second}\ g = \mathit{second}\ g \ggg \mathit{first}\ f$
product $\mathit{init}\ i \star\!\star\!\star\ \mathit{init}\ j = \mathit{init}\ (i, j)$

Based on the abstract arrow laws, an important property of CCA is that they enjoy a canonical form called *Causal Commutative Normal Form* (CCNF) that is either a pure arrow of the form $\mathit{arr}\ f$, or $\mathit{loop}\ (\mathit{arr}\ f \ggg \mathit{second}\ (\mathit{second}\ (\mathit{init}\ i)))$ for some initial state i and a pure function f. Furthermore if we relax the condition and allow recursions in the pure function, we end up with an *optimized CCNF* of the form $\mathit{loop}\ (\mathit{arr}\ g \ggg \mathit{second}\ (\mathit{init}\ i))$. For example, the arrow program for damped oscillator is translated to the optimized CCNF below:

```
loop (arr g ⋙ second (init i))
  where i = (y_0, y_1)
        g (_, (y, y')) = let y'' = −c ∗ y' − y
                         in  (y, (y', y''))
```

This kind of normalization can be seen as a stated compilation that turns an arrow program into a pair (i, g) where

- The stat i is a nested tuple that can be viewed as a vector since all states in our ODEs are of the same numerical types.
- The pure function g computes the derivative the state vector.

With this result in mind, we implement a new sampling function as follows:

```
class VectorSpace v a where
  (∗^) :: v → a → a
instance Num a ⇒ VectorSpace a a where
  x ∗^ y = x ∗ y
instance (VectorSpace v a, VectorSpace v b) ⇒ VectorSpace v (a, b) where
  k ∗^ (x, y) = (k ∗^ x, k ∗^ y)
instance (Num a, Num b) ⇒ Num (a, b) where
  negate  (x, y) = (negate x, negate y)
  (x, y) + (u, v) = (x + u, y + v)
  (x, y) ∗  (u, v) = (x ∗ u, y ∗ v)
  ...
euler         :: (VectorSpace v a, Num a) ⇒ v → a → a → a
euler h i i'  = i + h ∗^ i'
sample        :: (VectorSpace v a, Num a) ⇒ v → (a, ((), a) → (b, a)) → [b]
sample h (i, f) = aux i
      where aux i = x : aux j
              where (x, i') = f ((), i)
                    j       = euler h i i'
```

The *VectorSpace* class captures state vectors with a scalar multiplication operator $*\hat{}$, and also regains the homogeneous type required by *euler*. Such tuples are made instances of the *Num* class, where arithmetic operators are overloaded point-wise. The *sample* function then takes the tuple (i, g) we obtain from the optimized CCNF of an arrow program, uses function g to calculate the derivative of i, and computes its next step value using *euler*.

Now it becomes even clearer that there is no leak because only the state vector is updated during the repeated sampling, while the pure function remains unchanged. In addition, it runs very fast when compiled with GHC thanks to the normalization of CCA.

6 Benchmark

We compare the DSL performance of the tagged solution, the ODE arrow and CCA-based staged compilation by running ODE examples listed in Figure 1 and Figure 4. We do not consider the very first DSL and the HOAS version because they both have space leaks, and neither do we include results from the memoized HOAS version since it is always slower than the tagged DSL. The benchmarks were run on an Intel Pentium 4 machine running a 32-bit Linux OS. All programs are compiled to compute 10^5 samples using GHC 6.10.4 with compilation flag `-O2 -fvia-C`. The results are given in Figure 6, where all numbers are speed-up ratios measured in CPU time normalized to the speed of the first column. We make the following observations:

1. As the ODE gets more complex (from sine to oscillator, and to Lorenz), the tagged version becomes slower since it incurs more overhead interpreting the DSL, as well as remembering and comparing visited tags.
2. The Arrow version is slower than the tagged version for simpler ODEs, which is attributed to the overhead of interpreting the arrow combinators.
3. The CCA version is orders of magnitude faster since it is free of all arrow and arrow notation overhead. The intermediate *Core* program generated by GHC also confirms that the CCA optimization leads to very efficient target code in a tight loop.

7 Discussion

Before discussing the sharing problem in general, one may ask why we take the long road implementing a DSL for ODEs, when they can be directly represented

	Tagged	Arrow	CCA
Sine wave	1	0.31	14.06
Damped oscillator	1	0.75	35.48
Lorenz attractor	1	1.79	48.79

Fig. 6. Benchmark of DSLs for ODE (normalized speed)

in Haskell as a function that computes derivatives. For example, the damped oscillator ODE in Figure 1 can be described as follows:

$$\begin{aligned} f\ (y, y') = &\ \mathbf{let}\ y'' = -c * y' - y \\ &\ \mathbf{in}\ \ (y', y'') \end{aligned}$$

Coupled with a set of initial values (y_0, y_1), we have a pair from which numerical solutions to the ODE can be computed. A major drawback, however, is that such a pair is at too low level because it is unable to:

1. express the function represented by an ODE as a single value;
2. express compositions such as $y * y$ where y is defined by the above ODE;
3. make room for new extensions.

The lack of abstraction renders such a direct representation a poor choice for a DSL. Moreover, the purpose of this paper is not to solve differential equations, but to explore the design space of embedded DSLs that preserves sharing of computation. It is also worth noting that our staged compilation through CCA yields a similar pair of function and state.

Memoization [12] caches previous computation results and later re-uses instead of re-computes them. A generic *memo* function builds an internal lookup table that may interfere with garbage collection, and the prompt release of cached data is critical to the success of this technique.

The sharing problem discussed in this paper is of course not new. A majority of efforts have focused on detecting cycles and properly representing them. O'Donnell [14] uses integer tags for explicit labelling, while Claessen and Sands [3] suggest a non-conservative extension using references. Gill [7] introduces type-safe observable sharing using stable names within the IO monad. These techniques usually translate a lazy cyclic structure into an equivalent graph representation, but are inefficient at handing updates.

Introducing variable bindings to denote sharing or recursion in an algebraic data type is not new either. Fegaras and Sheard [5] adopt HOAS, while Ghani et al. [6] employ de Bruijn indices in a nested data type [1].

Historically the normal order reduction of a combinator program is known to preserve sharing in a similar way to lazy (call-by-need) evaluation [16], but such a style has rarely been used to represent sharing or cycles in algebraic data types despite having less overhead than both variable bindings and de Bruijn indices. The arrow abstraction gives rise to a rich algebra in a combinator style, which makes it a suitable candidate for traversals and updates, as well as transformations using the set of arrow laws. The abstract properties of arrows are powerful enough that they lead to the discovery of a normal form for CCA [11], and a staged compilation technique that eliminates all interpretive overhead.

Acknowledgements This research was supported in part by NSF grants CCF-0811665 and CNS-0720682, and by a grant from Microsoft Research.

Bibliography

[1] Richard S. Bird and Ross Paterson. de bruijn notation as a nested datatype. *J. Funct. Program.*, 9(1):77–91, 1999. ISSN 0956-7968.

[2] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in haskell. In *ICFP '98: International Conference on Functional Programming*, pages 174–184. ACM Press, 1998.

[3] Koen Claessen and David Sands. Observable sharing for functional circuit description. In *Asian Computing Science Conference*, pages 62–73. Springer Verlag, 1999.

[4] Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In *Proc. of the 2001 ACM SIGPLAN Haskell Workshop*. ACM Press, 2001.

[5] Leonidas Fegaras and Tim Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In *POPL'96: Proc. of the 17th symposium on Principles of programming languages*, pages 284–294. ACM Press, 1996.

[6] N. Ghani, M. Hamana, T. Uustalu, and V. Vene. Representing cyclic structures as nested datatypes. In *Proc. of 7th Symposium on Trends in Functional Programming (TFP 2006)*, pages 173–188, 2006.

[7] Andy Gill. Type-safe observable sharing in Haskell. In *Proc. of the 2009 ACM SIGPLAN Haskell Symposium*. ACM Press, 2009.

[8] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, 2000.

[9] Jerzy Karczmarczuk. Functional differentiation of computer programs. In *ICFP '98: International Conference on Functional Programming*, pages 195–203, 1998.

[10] Hai Liu and Paul Hudak. Plugging a space leak with an arrow. *Electronic Notes in Theoretical Computer Science*, 193:29–45, 2007.

[11] Hai Liu, Eric Cheng, and Paul Hudak. Causal commutative arrows and their optimization. In *ICFP '09: International Conference on Functional Programming*, pages 35–46. ACM Press, 2009.

[12] Donald Michie. Memo functions and machine learning. *Nature*, 218(5136): 19–22, 1968.

[13] Henrik Nilsson, Antony Courtney, and John Peterson. Functional Reactive Programming, continued. In *Proc. of ACM SIGPLAN 2002 Haskell Workshop*. ACM Press, 2002.

[14] John T. O'Donnell. Generating netlists from executable circuit specifications in a pure functional language. In *Functional Programming Glasgow, Springer-Verlag Workshops in Computing*, pages 178–194. Springer-Verlag, 1992.

[15] Ross Paterson. A new notation for arrows. In *ICFP '01: International Conference on Functional Programming*, pages 229–240. ACM Press, 2001.

[16] D.A. Turner. A new implementation technique for applicative languages. *Software-Practice and Experience*, 9:31–49, 1979.