# How to Twist Pointers without Breaking Them

Satvik Chauhan [*]
Google
satvik@google.com

Piyush P. Kurur
Indian Institute Of Technology Kanpur
Kanpur, UP 208016, India
ppk@cse.iitk.ac.in

Brent A. Yorgey
Hendrix College
Conway, Arkansas, USA
yorgey@hendrix.edu

## Abstract

Using the theory of monoids and monoid actions, we give a unified framework that handles three common pointer manipulation tasks, namely, data serialisation, deserialisation, and memory allocation. Our main theoretical contribution is the formulation of the notion of a *twisted functor*, a generalisation of the semi-direct product construction for monoids. We show that semi-direct products and twisted functors are particularly well suited as an abstraction for many pointer manipulation tasks.

We describe the implementation of these abstractions in the context of a cryptographic library for Haskell. Twisted functors allow us to abstract all pointer arithmetic and size calculations into a few lines of code, significantly reducing the opportunities for buffer overflows.

***Categories and Subject Descriptors*** D.1.1 [*Programming techniques*]: Applicative (Functional) Programming

***General Terms*** Languages, Security

***Keywords*** monoid, monoid action, applicative functor, semi-direct product, pointer

## 1. Introduction

Pointers as an abstraction for memory locations is a powerful idea which is also, unfortunately, a fertile source of bugs. It is difficult to catch illegal access to memory referenced by pointers, leading, for example, to dreaded buffer overflows. The work that we describe in this article grew out of the need to handle pointers carefully in `raaz` (Chauhan and Kurur), a cryptographic network library written in Haskell. An important goal in the design of this library is to leverage the type safety of Haskell, whenever possible, to avoid common bugs.

One might question the need for manual memory allocation or direct pointer manipulation in a high-level language like Haskell. Indeed, one should avoid such low level code for most programming tasks. Cryptographic implementations, however, are different. The low-level primitives are often implemented using foreign function calls to C or even assembly. While performance is the primary motivation for including such low-level code, it is also a necessary security measure, since otherwise compilers can introduce optimisations that enable various side-channel attacks.

Calling such C or assembly code via Haskell's foreign function interface requires marshalling data back and forth across the boundary, explicitly allocating buffers and using pointer manipulation to read and write data. Explicit memory allocation is also necessary for storing sensitive data, as Haskell's runtime can otherwise leak information by moving data during garbage collection.

Our main contributions are the following:

- We explain the *semi-direct product* of monoids (Section 4.2) and give several examples of their use (Section 4.3). Though well-known in mathematical contexts, semi-direct products are perhaps not as well-known as they ought to be among functional programmers.
- We formalise *twisted functors* (Section 5), a generalisation of semi-direct products. Despite seeming somewhat "obvious" in hindsight, twisted functors are, to our knowledge, novel (a discussion of related work can be found in Section 10).
- We use monoid actions, semi-direct products, and twisted functors to develop a unified framework capable of abstracting three common tasks using pointers, namely, data serialisation (Sections 2.2 and 7), deserialisation (Sections 2.3 and 7), and memory allocation (Section 8).

The advantage of our interface is that the only fragment of code that performs any pointer arithmetic is the instance declaration of the type class *Action* given in Section 7 (about three lines of code). Twisted functors automatically take care of all the offset and size calculations required to perform common pointer manipulation tasks, considerably shrinking the attack surface for our library.

More importantly, the twisted functor construction is mathematically well motivated (see Section 9 for a category theoretic motivation). We believe that twisted functors may have other applications to programming as well.

## 2. Why the Twist?

The goal of this section is to introduce the basic patterns behind the twisted functor construction, using pointer manipulation tasks as examples. We begin by developing a *Monoid* instance to help with data serialisation, which is built out of two other monoids—but instead of the usual product construction, where the two monoids are combined in parallel, we need a more complicated construction where the two monoids interact, known as the *semi-direct product* of monoids. We then turn to deserialisation and show how it follows a similar pattern, but generalised to applicative functors instead of monoids. This generalisation of a semi-direct product is what we call a *twisted functor*.

[*] Work done when the author was at the Department of Computer Science and Engineering, Indian Institute of Technology Kanpur

### 2.1 Monoids and Applicative Functors

We first quickly review a few preliminary concepts. Recall that a *monoid* $(m, \diamond, \varepsilon)$ is a type $m$ together with an associative operation $\diamond$ which has an identity element $\varepsilon$:

```
class Monoid m where
  ε   :: m
  (◇) :: m → m → m
```

The requirements on $\varepsilon$ and $(\diamond)$ can be summarized by the following laws:

$$\begin{array}{lll} \textbf{identity:} & \varepsilon \diamond x & = x \\ & x \diamond \varepsilon & = x \\ \textbf{associativity:} & a \diamond (b \diamond c) & = (a \diamond b) \diamond c \end{array}$$

A *commutative* monoid additionally satisfies the law $a \diamond b = b \diamond a$ for all $a$ and $b$. We will use the empty *Commutative* class to tag monoids which are also commutative:

```
class Monoid m ⇒ Commutative m
```

We assume a basic familiarity with monoids on the part of the reader. We also assume a basic familiarity with *applicative functors* (McBride and Paterson 2008), represented in Haskell via the following standard type class:

```
class Functor f ⇒ Applicative f where
  pure  :: a → f a
  (⟨∗⟩) :: f (a → b) → f a → f b
```

The laws for *Applicative* are less important at this point, and will be discussed later in Section 3.

Although it may not be obvious from the above presentations, an important point is that *Applicative can be seen as a generalisation of Monoid*. We will have more to say about this in Section 3; for now, a reader unfamiliar with this fact may like to use the following examples to contemplate the relationship between *Monoid* and *Applicative*.

### 2.2 Data Serialisation

As a starting point for discussing data serialisation, consider the following type:

```
newtype WriteAction = WA (Pointer → IO ())
```

The idea is that a *WriteAction* takes a pointer as input and writes some bytes to memory beginning at the referenced location. The *Pointer* type represents a generic pointer which can point to any particular byte in memory. In reality one would use a type such as *Ptr Word*, but the specific implementation details are not important.

We can naturally make *WriteAction* into a monoid as follows:

```
instance Monoid WriteAction where
  ε                = WA $ const (return ())
  WA w₁ ◇ WA w₂ = WA $ λptr → w₁ ptr ≫ w₂ ptr
```

That is, the product $w_1 \diamond w_2$ of two write actions performs the write action $w_1$ followed by the write action $w_2$ on the *same* input pointer; the identity element is the function that ignores the pointer and does nothing. However, this is useless as an abstraction for serialisation, since $w_2$ overwrites bytes written by the previous action $w_1$.

The main problem with *WriteAction* is that we have no way to know how many bytes are written. To remedy this, consider the pair type (*WriteAction*, *Sum Int*). An element $(w, n)$ of this type is interpreted as a write action $w$ tagged with the number of bytes that will be written by $w$. We wrap *Int* inside *Sum* so it has an additive monoid structure.

This pair type (*WriteAction*, *Sum Int*) automatically has a *Monoid* instance which works componentwise, that is,

$$(w_1, n_1) \diamond (w_2, n_2) = (w_1 \diamond w_2, n_1 + n_2).$$

However, this is not much better. Even though the second component is indeed doing its job of keeping track of the total number of bytes written, as the first component we still have $w_1 \diamond w_2$, where $w_2$ overwrites bytes previously written by $w_1$. We want a different monoidal structure for (*WriteAction*, *Sum Int*); in particular, the tracked sizes should affect the way the write actions are composed.

Intuitively, after running $w_1$, we should first shift the input pointer by $n_1$ bytes before running $w_2$, ensuring that $w_1$ and $w_2$ write their bytes sequentially. That is, we want something like

$$(w_1, n_1) \diamond (w_2, n_2) = (w_1 \diamond (n_1 \bullet w_2), n_1 + n_2),$$

where

```
Sum n₁ • WA w₂ = WA $ λptr → w₂ (shiftPtr n₁ ptr)
```

denotes the *WriteAction* which first shifts the input pointer by $n_1$ bytes before running $w_2$ (assuming a suitable function *shiftPtr* :: *Int* → *Pointer* → *Pointer*). As we will show in more detail later (Section 4.2), this does indeed define a valid monoid, known as the *semi-direct product* of *WriteAction* and *Sum Int*.

### 2.3 Deserialisation

Consider the opposite problem of *deserialising* a Haskell value from a buffer of bytes. We now build a rudimentary framework which exposes an applicative functor based interface (McBride and Paterson 2008; Swierstra and Duponcheel 1996), beginning with the following type:

```
newtype ParseAction a = PA (Pointer → IO a)
```

First, we can make a straightforward *Applicative* instance for *ParseAction*:

```
instance Applicative ParseAction where
  pure             = PA ∘ const ∘ return
  PA pf ⟨∗⟩ PA px = PA $ λptr → pf ptr ⟨∗⟩ px ptr
```

That is, *pure* just returns the value without reading any bytes, and *pf* ⟨∗⟩ *px* on input *ptr* parses a function from *ptr* using *pf* and applies it to the value parsed from *ptr* via *px*.

As an aside, we note that this is just the composition of the *Applicative* instance for *IO* with the instance for ((→) *Pointer*); we could get this *Applicative* instance for free if we defined

```
type ParseAction′ = Compose ((→) Pointer) IO,
```

but for our purposes the extra syntactic noise of dealing with *Compose* would outweigh the benefit. Note also that defining

```
type ParseAction′ a = Pointer → IO a
```

does not work at all, since the standard *Applicative* instance for ((→) *Pointer*) cannot take the *IO* into account.

However, we now face a similar problem as in the previous section: by default, *pf* ⟨∗⟩ *px* parses both the function and value from the *same* location. Instead, we would like *px* to pick up parsing where *pf* left off. Again, the underlying reason is that we do not know how many bytes *pf* will read.

The solution is a construction analogous to that of the previous section, but carried out for an applicative functor instead of a monoid. We make a *Parser* type by pairing a *ParseAction* with a tag tracking the number of bytes read:

```
data Parser a = P (ParseAction a) (Sum Int)
```

Again, there is a standard *Applicative* instance we could make for *Parser* which just combines *ParseAction* and *Sum Int* values separately, but we don't want that instance. Instead, we use the following instance:

```
instance Applicative Parser where
  pure a                 = P (pure a)              0
  P pf n₁ ⟨∗⟩ P px n₂ = P (pf ⟨∗⟩ (n₁ ⊙ px)) (n₁ + n₂)
```

Here $n_1 \odot px$ is the operation, analogous to $n \bullet ptr$ from the previous section, that first shifts the pointer by $n_1$ bytes and then parses using $px$. This construction is essentially what we call a *twisted functor*; later, in Section 5.2, we will define twisted functors more formally and show how the above $\mathit{Applicative}$ instance can be built automatically out of the $\mathit{Sum}\ \mathit{Int}$ monoid and its *action* on the $\mathit{ParseAction}$ applicative functor.

The $\mathit{Applicative}$ interface allows us to define complicated parsers in terms of simpler ones much like other parser combinator libraries such as `attoparsec` (O'Sullivan). For example, consider parsing a tuple that consists of a 64-bit word followed by a 32-bit word. We can express such a parser in terms of the parsers for its components as follows:

$$\mathit{parseTuple} = (,)\ \langle\$\rangle\ \mathit{parse64}\ \langle*\rangle\ \mathit{parse32}$$

Assuming $\mathit{parse64}$ and $\mathit{parse32}$ have been defined properly, this parser will read a 64-bit word from whatever memory location it is given, and then automatically read another 32-bit word from the location immediately following.

## 3. Applicative Functors as Generalised Monoids

The close relationship of applicative functors and monoids is somewhat obscured by the usual presentation of $\mathit{Applicative}$. The $\mathit{Applicative}$ laws are essentially geared towards their use as a generalized $\mathit{zip}$ and have nothing obviously in common with the $\mathit{Monoid}$ laws. Consider instead the following type class (McBride and Paterson 2008, Section 7):

$$\begin{aligned}&\textbf{class}\ \mathit{Functor}\ f \Rightarrow \mathit{Monoidal}\ f\ \textbf{where}\\ &\quad \mathit{unit} :: f\ ()\\ &\quad (\star)\ :: f\ a \to f\ b \to f\ (a, b)\end{aligned}$$

Instances of $\mathit{Monoidal}$ are required to satisfy the following laws, which in essence say that the operator $\star$ is associative with $\mathit{unit}$ as an identity:

$$\begin{aligned}&\textbf{identity:} && \mathit{unit} \star v \cong v\\ & && u \star \mathit{unit} \cong u\\ &\textbf{associativity:} && u \star (v \star w) \cong (u \star v) \star w\end{aligned}$$

where $\cong$ is the isomorphism of types generated by $((), a) \cong a \cong (a, ())$ and $(a, (b, c)) \cong ((a, b), c)$.

$\mathit{Monoidal}$ clearly forms a sort of "type-indexed" generalisation of $\mathit{Monoid}$, where the monoid structure is reflected at the level of types by the unit and product types (which themselves form a type-level monoid structure up to isomorphism). The $\mathit{Monoidal}$ laws are also clear generalisations of the $\mathit{Monoid}$ laws, with two laws for left and right identity and one law for associativity.

In fact, the $\mathit{Monoidal}$ type class can be seen as an alternative presentation of $\mathit{Applicative}$. It is not hard to implement $\mathit{pure}$ and $(\langle*\rangle)$ in terms of $\mathit{fmap}$, $\mathit{unit}$, and $(\star)$, and vice versa:

$$\begin{aligned}\mathit{pure}\ x &= \mathit{const}\ x\ \langle\$\rangle\ \mathit{unit}\\ u\ \langle*\rangle\ v &= \mathit{uncurry}\ (\$)\ \langle\$\rangle\ (u \star v)\\ \mathit{fmap}\ f\ x &= \mathit{pure}\ f\ \langle*\rangle\ x\\ \mathit{unit} &= \mathit{pure}\ ()\\ u \star v &= \mathit{pure}\ (,)\ \langle*\rangle\ u\ \langle*\rangle\ v\end{aligned}$$

With these definitions, it is also true (though less obvious) that the $\mathit{Monoidal}$ and $\mathit{Functor}$ laws together imply the $\mathit{Applicative}$ laws, and vice versa. See McBride and Paterson (2008, Section 7) for details.

Both because of the more evident connection to $\mathit{Monoid}$, and because of the more intuitive proof obligations, we will present the rest of the paper in terms of the $\mathit{Monoidal}$ class, but converting our results to use $\mathit{Applicative}$ is straightforward.

## 4. Monoid Actions and Semi-Direct Products

We first study the *semi-direct product* of two monoids, a well-known algebraic construction which deserves to be better known among functional programmers.

### 4.1 Monoid Actions

Recall the notion of a *monoid action* (see also Yorgey (2012)): given a monoid $m$ and an arbitrary type $a$, a *left action*[1] of $m$ on $a$ is a function $(\bullet) : m \to a \to a$ by which the elements of $m$ behave like transformations on the type $a$. Left actions are captured by the type class $\mathit{Action}$ (defined in $\mathit{Data.Monoid.Action}$ from the `monoid-extras` package (Yorgey)):

$$\begin{aligned}&\textbf{class}\ \mathit{Monoid}\ m \Rightarrow \mathit{Action}\ m\ a\ \textbf{where}\\ &\quad (\bullet) :: m \to a \to a\end{aligned}$$

Every instance of $\mathit{Action}$ is required to satisfy the following laws:

$$\begin{aligned}&\textbf{identity:} && \varepsilon \bullet a = a\\ &\textbf{composition:} && (m_1 \diamond m_2) \bullet a = m_1 \bullet (m_2 \bullet a)\end{aligned}$$

A left action associates each element of $m$ to a transformation on $a$, that is, a function of type $a \to a$. The laws specify how the monoid structure of $m$ is reflected in these transformations; for example, the transformation associated to the product $m_1 \diamond m_2$ must be the composition of the respective transformations. Notice that the type $a \to a$ is itself a monoid, where the identity function $\mathit{id}$ is the unit $\varepsilon$ and the monoid operation is function composition. The laws can thus be seen as requiring a left action to be a *homomorphism* from the monoid $m$ to the monoid $a \to a$. This is also brought out much more clearly by equivalent point-free versions of the laws:

$$\begin{aligned}&\textbf{identity:} && (\varepsilon \bullet) = \mathit{id}\\ &\textbf{composition:} && ((m_1 \diamond m_2)\bullet) = (m_1\bullet) \circ (m_2\bullet)\end{aligned}$$

So far, we have been discussing monoid actions on arbitrary types. To define semi-direct product, we need to look at monoids acting on other monoids. In such situations we insist that the monoid action of $m$ on $a$ preserve the monoid structure of $a$, as captured by the following additional laws:

$$\begin{aligned}&\textbf{annihilation:} && m \bullet \varepsilon = \varepsilon\\ &\textbf{distributivity:} && m \bullet (a_1 \diamond a_2) = (m \bullet a_1) \diamond (m \bullet a_2)\end{aligned}$$

We call an action satisfying these additional laws a *distributive* action. Recall that when a monoid $m$ acts on a type $a$, elements of the type $m$ can be seen as elements of the type $a \to a$. A distributive action is then precisely one for which the associated functions $a \to a$ are themselves monoid homomorphisms. We capture this constraint by the type class $\mathit{Distributive}$:

$$\textbf{class}\ (\mathit{Action}\ m\ a, \mathit{Monoid}\ a) \Rightarrow \mathit{Distributive}\ m\ a$$

This class has no methods, but serves only to remind us of the additional laws.

### 4.2 Semi-Direct Products

Let $m$ be a monoid acting distributively on a type $a$, which itself is a monoid. Then the *semi-direct product* $a \rtimes m$ is structurally just the product type $(a, m)$, but under the monoid operation

$$(a_1, m_1) \diamond (a_2, m_2) = (a_1 \diamond (m_1 \bullet a_2), m_1 \diamond m_2).$$

This is similar to the usual product of monoids (where $(a_1, m_1) \diamond (a_2, m_2)$ simply yields $(a_1 \diamond a_2, m_1 \diamond m_2)$), but with a "twist" arising from the action of $m$ on $a$: the value $m_1$ in the first pair acts on the $a_2$ from the second pair before it is combined with $a_1$.

[1] In this paper, by an *action*, we always mean a left action. It is possible to define right actions analogously, but we do not need them.

Implementing semi-direct products is straightforward:

$$
\begin{array}{l}
\textbf{data}\ (\rtimes)\ a\ m = a :\rtimes m \\
\mathit{unSemi} :: (a \rtimes m)\ \to (a, m) \\
\mathit{unSemi}\quad (a :\rtimes m)\ = (a, m) \\
\textbf{instance}\ \mathit{Distributive}\ m\ a \Rightarrow \mathit{Monoid}\ (a \rtimes m)\ \textbf{where} \\
\quad \varepsilon \qquad\qquad\qquad\qquad\quad = \varepsilon :\rtimes \varepsilon \\
\quad (a_1 :\rtimes m_1) \diamond (a_2 :\rtimes m_2) = (a_1 \diamond (m_1 \bullet a_2)) :\rtimes (m_1 \diamond m_2)
\end{array}
$$

Semi-direct products of monoids are included in the `monoid-extras` package on Hackage (as of version 0.4.1).

We can prove that $a \rtimes m$ is indeed a monoid using the laws for a distributive action.

*Proof.* First, we show that $\varepsilon_a :\rtimes \varepsilon_m$ is both a left and right identity:

$$
\begin{array}{ll}
 & (\varepsilon_a :\rtimes \varepsilon_m) \diamond (a :\rtimes m) \\
= & \qquad\qquad\qquad \{\ \text{defn}\ \} \\
 & (\varepsilon_a \diamond (\varepsilon_m \bullet a) :\rtimes \varepsilon_m \diamond m) \\
= & \qquad\qquad\qquad \{\ \varepsilon_a, \varepsilon_m \text{ are identites}\ \} \\
 & (\varepsilon_m \bullet a :\rtimes m) \\
= & \qquad\qquad\qquad \{\ \text{identity law}\ \} \\
 & a :\rtimes m
\end{array}
$$

$$
\begin{array}{ll}
 & (a :\rtimes m) \diamond (\varepsilon_a :\rtimes \varepsilon_m) \\
= & \qquad\qquad\qquad \{\ \text{defn}\ \} \\
 & a \diamond (m \bullet \varepsilon_a) :\rtimes m \diamond \varepsilon_m \\
= & \qquad\qquad\qquad \{\ \varepsilon_m \text{ is identity}\ \} \\
 & a \diamond (m \bullet \varepsilon_a) :\rtimes m \\
= & \qquad\qquad\qquad \{\ \text{annihilation law}\ \} \\
 & a \diamond \varepsilon_a :\rtimes m \\
= & \qquad\qquad\qquad \{\ \varepsilon_a \text{ is identity}\ \} \\
 & a :\rtimes m
\end{array}
$$

So far, we have used the identity and annihilation laws for the distributive action of $m$ on $a$. Next, we prove associativity of the monoid operation, which relies on the other two laws, composition and distributivity:

$$
\begin{array}{ll}
 & ((a_1 :\rtimes m_1) \diamond (a_2 :\rtimes m_2)) \diamond (a_3 :\rtimes m_3) \\
= & \qquad\qquad\qquad \{\ \text{defn}\ \} \\
 & (a_1 \diamond (m_1 \bullet a_2) :\rtimes m_1 \diamond m_2) \diamond (a_3 :\rtimes m_3) \\
= & \qquad\qquad\qquad \{\ \text{defn}\ \} \\
 & (a_1 \diamond (m_1 \bullet a_2)) \diamond ((m_1 \diamond m_2) \bullet a_3) :\rtimes (m_1 \diamond m_2) \diamond m_3 \\
= & \qquad\qquad\qquad \{\ \text{associativity of } \diamond\ \} \\
 & a_1 \diamond ((m_1 \bullet a_2) \diamond ((m_1 \diamond m_2) \bullet a_3)) :\rtimes m_1 \diamond (m_2 \diamond m_3) \\
= & \qquad\qquad\qquad \{\ \text{composition}\ \} \\
 & a_1 \diamond ((m_1 \bullet a_2) \diamond (m_1 \bullet (m_2 \bullet a_3))) :\rtimes m_1 \diamond (m_2 \diamond m_3) \\
= & \qquad\qquad\qquad \{\ \text{distributivity}\ \} \\
 & a_1 \diamond (m_1 \bullet (a_2 \diamond (m_2 \bullet a_3))) :\rtimes m_1 \diamond (m_2 \diamond m_3) \\
= & \qquad\qquad\qquad \{\ \text{defn}\ \} \\
 & (a_1 :\rtimes m_1) \diamond (a_2 \diamond (m_2 \bullet a_3) :\rtimes m_2 \diamond m_3) \\
= & \qquad\qquad\qquad \{\ \text{defn}\ \} \\
 & (a_1 :\rtimes m_1) \diamond ((a_2 :\rtimes m_2) \diamond (a_3 :\rtimes m_3)) \quad \square
\end{array}
$$

So we see that a distributive action is precisely what is needed to define the semi-direct product.

### 4.3 Examples

Semi-direct products arise naturally in many situations; here are a few examples of semi-direct products in the context of functional programming.

***Pointer-based write actions*** Our first example is the monoid of length-tagged write actions that we sketched in Section 2.2. Recall that the operation for length-tagged write actions is

$$(w_1, n_1) \diamond (w_2, n_2) = (w_1 \diamond (n_1 \bullet w_2), n_1 + n_2),$$

where $n_1 \bullet w_2$ is the write action which shifts the input pointer by $n_1$ before writing. Now that we have seen the formal definitions of monoid actions and semi-direct products, we can see that this is precisely the semi-direct product of $\mathit{WriteAction}$ and $\mathit{Sum\ Int}$. The exact implementation of the type $\mathit{Write}$ in our library is slightly more general; we describe it in detail in Section 7.

***Computing $\mathit{GEN}$- and $\mathit{KILL}$-sets*** Consider a simple *straight line program* which consists of a list of assignment instructions of the form $x := f(y_1, ..., y_k)$, where $x$ and the $y_i$ are variables and $f$ is a built-in function. A standard computational task that is used during data-flow analysis in a compiler is to compute, given a straight line program $p$, the following sets:

**$\mathit{GEN}$-set:** The $\mathit{GEN}$-set is the set of variables that are *used before they are assigned* in $p$. For example, for the program

$$
\begin{array}{l}
y := g() \\
x := y + z \\
z := x - w
\end{array}
$$

the $\mathit{GEN}$-set is $\{z, w\}$, both of which are referenced on the right-hand side of an assignment before they appear on the left of an assignment (in fact, $w$ never appears on the left of an assignment in this program).

**$\mathit{KILL}$-set:** The set of variables that are *assigned* in $p$. The $\mathit{KILL}$-set for the above example program would be $\{x, y, z\}$.

If the program $p_1; p_2$ is defined as the concatnation of the programs $p_1$ and $p_2$, the question becomes how to compositionally compute the $\mathit{GEN}$- and $\mathit{KILL}$-sets of $p_1; p_2$ from those of $p_1$ and $p_2$. We show that semi-direct products can be used for this purpose.

Consider the monoid $M$ of sets of variables, with union as the monoid operation. $M$ acts on itself via $s \bullet t = t \setminus s$, that is, $s \bullet t$ is the set $t$ with the variables in $s$ removed. Since the empty set is the identity action, and removing $s_1$ then removing $s_2$ is the same as removing $s_1 \cup s_2$, this is a valid action. Moreover, it is distributive: removing variables from the empty set has no effect, and $(t_1 \cup t_2) \setminus s = (t_1 \setminus s) \cup (t_2 \setminus s)$.

The $\mathit{KILL}$-set of $p_1; p_2$ is the set of variables assigned in $p_1; p_2$, which is just the union of the variables assigned in $p_1$ and $p_2$. The $\mathit{GEN}$-set of $p_1; p_2$, that is, variables used before being defined, is the union of those used before being assigned in either $p_1$ or $p_2$, *except* for variables in the $\mathit{GEN}$-set of $p_2$ but in the $\mathit{KILL}$-set of $p_1$: such variables are now assigned before being used.

More formally, if $(g_i, k_i)$ is the ordered pair of $\mathit{GEN}$- and $\mathit{KILL}$-sets for $p_i$ $(i = 1, 2)$, then the $\mathit{GEN}$- and $\mathit{KILL}$-sets associated with the program $p_1;\ p_2$ are given by:

$$
\begin{array}{rcl}
(g, k) & = & (g_1 \cup (g_2 \setminus k_1), k_1 \cup k_2) \\
 & = & (g_1 \diamond (k_1 \bullet g_2), k_1 \diamond\ k_2)
\end{array}
$$

From a computational perspective it is therefore natural to consider the pair of $\mathit{GEN}$- and $\mathit{KILL}$-sets as elements of the semi-direct product $M \rtimes M$ — computing them for a straight line program, which is just a list of instructions, amounts to computing them for each instruction in the program followed by applying $\mathit{mconcat}$.

***Tangent-matched path joins*** The `diagrams` library (Yates and Yorgey 2015; Yorgey 2012) defines a notion of *trails*, which represent *translation-invariant* motions through space. That is, a trail does not have a definite start and end location, but rather expresses a particular movement relative to any given starting location, much

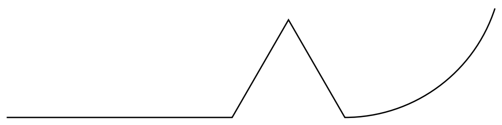

**Figure 1.** An example trail

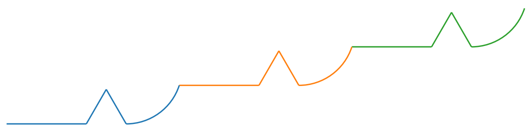

**Figure 2.** The trail concatenated three times

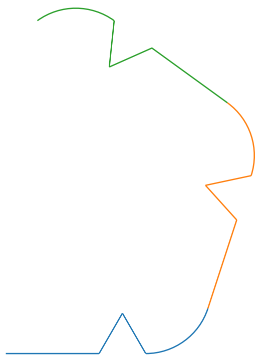

**Figure 3.** The trail concatenated three times, with tangent matching

as a vector has a magnitude and direction but no concrete location. Figure 1 shows an example trail; from any given starting point, the trail will trace out the shape shown (from left to right).

Trails have a natural *Monoid* instance with concatenation as the combining operation. For example, combining three copies of the trail shown in Figure 1 produces Figure 2.

Note, however, that each trail has a definite orientation which is unaffected by previous trails. For some applications, we might prefer an alternative *Monoid* instance where concatenated trails are rotated so the tangent vector at the end of each trail matches that at the start of the next. For example, again combining three copies of the trail shown above, but using this alternate *Monoid* instance, would produce the trail shown in Figure 3.

In fact, this alternate monoid can be elegantly built as a semi-direct product. The other monoid we need is that of angles under addition; angles act on trails by rotation. We can verify that this is a distributive action: rotating by an angle of 0 has no effect, and rotating by two angles in succession is the same as rotating by their sum; likewise, rotating the empty trail does nothing, and rotating a concatenation of two trails gives the same result as rotating them individually and then concatenating.

We therefore consider the semi-direct product $\mathit{Trail} \rtimes \mathit{Angle}$, where we maintain the invariant that the angle tagged onto a *Trail* represents the angle of its final tangent vector, that is, the difference between the positive $x$-axis and its ending tangent vector (the positive $x$-axis is a conventional yet arbitrary choice; any consistent reference direction will do). In that case, the monoid operation for $\mathit{Trail} \rtimes \mathit{Angle}$ does something like what we want:

$$(t_1 :\rtimes \theta_1) \diamond (t_2 :\rtimes \theta_2) = t_1 \diamond (\theta_1 \bullet t_2) :\rtimes \theta_1 + \theta_2$$

$t_1 \diamond (\theta_1 \bullet t_2)$ concatenates $t_1$ and $t_2$, but not before rotating $t_2$ to match the amount of rotation introduced by $t_1$. We just need a function to appropriately create a value of the semi-direct product given a *Trail*: it measures the amount of rotation from the beginning to the end of the trail, and also rotates the trail so its beginning direction is along the positive x-axis. This normalization step is necessary so that rotating the trails later will cause them to properly align.

This monoid for trails is not yet included in the `diagrams` library, but is planned for addition soon.

## 5. Twisted Functors

We now turn to the main theoretical contribution of the article, namely *twisted functors*, a generalisation of semi-direct product.

### 5.1 Monoids Acting on Functors

We have seen that the semi-direct product is a way to construct a monoid out of two monoids, with one acting on the other. The *twisted functor* is the analogous notion where one of the monoids (the one being acted upon) is generalised to a monoidal (applicative) functor.

We first formalise the action of monoids on functors. Intuitively, for a monoid $m$ to act on the functor $f$, it should act in a uniform way on all the types $f\ a$. Therefore, we would like to assert a constraint something like `forall a. Action m (f a)`. Unfortunately, Haskell does not allow the use of universally quantified constraints and hence we need a new type class.

$$\begin{array}{l}\textbf{class}\ (\mathit{Monoid}\ m, \mathit{Functor}\ f) \Rightarrow \mathit{ActionF}\ m\ f\ \textbf{where} \\ \quad (\odot) :: m \to f\ a \to f\ a\end{array}$$

An instance of *ActionF* should satisfy the following laws:

$$\begin{array}{llll}\textbf{identity:} & \varepsilon & \odot\ fx & = fx \\ \textbf{composition:} & (a \diamond b) & \odot\ fx & = a \odot\ (b \odot fx) \\ \textbf{uniformity:} & m & \odot\ \mathit{fmap}\ f\ u & = \mathit{fmap}\ f\ (m \odot u)\end{array}$$

The first two laws state that $m$ acts on the left of each type $f\ a$. The last law says that the action of $m$ is in some sense uniform across the functor. We say that the monoid *m acts uniformly (on the left)* on the functor $f$. In fact, it is not strictly necessary to state the uniformity law, since it is obtained as a *free theorem* (Wadler 1989) from the type of $(\odot)$. That is, by parametricity, every (total) function with the type of $(\odot)$ automatically satisfies the uniformity law. (For a more detailed category theoretic explanation, see Section 9.) So we will generally omit the word "uniformly" and just say that *m acts on* the functor $f$.

When a monoid $m$ acts on a type $a$ which is itself a monoid, the interesting actions were the ones which satisfied the distributivity law. Something similar is required when we look at monoid actions on monoidal functors. The obvious generalisation (which thankfully turns out to be the right one) is the following:

$$\begin{array}{ll}\textbf{annihilation:} & m \odot \mathit{unit} \quad = \mathit{unit} \\ \textbf{distributivity:} & m \odot (u \star v) = (m \odot u) \star (m \odot v)\end{array}$$

It is worth stating what the equivalent laws would be for *Applicative*:

$$\begin{array}{ll}\textbf{stoicism:} & m \odot \mathit{pure}\ x \quad = \mathit{pure}\ x \\ \textbf{effectiveness:} & m \odot (f \langle * \rangle x) = (m \odot f) \langle * \rangle (m \odot x)\end{array}$$

Given the definitions of *Applicative* in terms of *Monoidal* and vice versa, and in the presence of the uniformity law, it is straightforward to prove the equivalence of these pairs of laws, which we leave as an exercise for the reader. If we think of the applicative functor as encapsulating computations with side effects, then the stoicism law prohibits the monoid from acting non-trivially on pure values, that is, "pure values are stoic". The effectiveness law says that the monoid action captured by the operator $\odot$ distributes across effects.

In any case, whether formulated in terms of *Monoidal* or *Applicative*, we call this style of action a *distributive action* of a monoid on a monoidal functor. This constraint is captured in Haskell using the following type class:

```
class (Monoidal f, ActionF m f) ⇒ DistributiveF m f
```

Again, this class has no methods, but only reminds us of the necessary laws.

### 5.1.1 Lifted Actions

An important class of distributive actions of a *commutative* monoid on a functor arises by what we call a lifted action. These lifted actions are precisely what we need when working with pointer functions.

Recall that the type $(\to)\ a$ is a monoidal functor, with a *Monoidal* instance defined by

```
instance Monoidal ((→) a) where
  unit  = const ()
  f ⋆ g = f △ g
```

where $f \mathbin{\triangle} g = \lambda x \to (f\ x, g\ x)$ is the function which pairs the results of $f$ and $g$ on the same argument (also written $f$ *&&&* $g$ in more ASCII-ish Haskell).

An action of the commutative monoid $m$ on the type $a$ can be *lifted* to the functor $(\to)\ a$ as defined below:

```
instance (  Commutative m, Action m a)
         ⇒ ActionF m ((→) a) where
  m ⊙ f = f ∘ (m•)
```

Notice that the function $(m\bullet)$ is an element of the type $a \to a$ and can be thought of as "shifting a value $a$ by $m$". The function $m \odot f$ therefore first shifts the input by $m$ and then applies $f$.

Although the action of $m$ on $a$ need not be distributive ($a$ is not even necessarily a monoid), the lifted action of $m$ on $((\to)\ a)$ is:

```
instance (  Commutative m, Action m a)
         ⇒ DistributiveF m ((→) a)
```

*Proof.* To prove that this is a valid distributive action, we must prove the four distributive action laws (identity, composition, annihilation, and distributivity).

- The identity law for the action of $m$ on $(\to)\ a$ follows from the same law for the action of $m$ on $a$:

  $$\varepsilon \odot f = f \circ (\varepsilon\bullet) = f \circ id = f$$

- The annihilation law does not even depend on any other laws, but follows directly from the definition of the *Monoidal* instance for $(\to)\ a$:

  $$m \odot unit = const\ () \circ (m\bullet) = const\ ()$$

- The distributivity law follows just by unfolding definitions, noting that function composition distributes over $(\triangle)$ from the right.

$$\begin{array}{ll}
m \odot (f \star g) & \\
= & \{\ \text{defn of } (\odot)\ \} \\
(f \star g) \circ (m\bullet) & \\
= & \{\ \text{defn of } (\star)\ \} \\
(f \mathbin{\triangle} g) \circ (m\bullet) & \\
= & \{\ (\circ) \text{ distributes over } (\triangle)\ \} \\
(f \circ (m\bullet)) \mathbin{\triangle} (g \circ (m\bullet)) & \\
= & \{\ \text{defn of } (\odot) \text{ and } (\star)\ \} \\
(m \odot f) \star (m \odot g) &
\end{array}$$

- The composition law turns out to be the most interesting, and depends crucially on both the commutativity of $m$ and the composition law for the action of $m$ on $a$.

$$\begin{array}{ll}
(a \diamond b) \odot f & \\
= & \{\ \text{defn of } (\odot)\ \} \\
f \circ ((a \diamond b)\bullet) & \\
= & \{\ \text{commutativity}\ \} \\
f \circ ((b \diamond a)\bullet) & \\
= & \{\ \text{composition, associativity of } (\circ)\ \} \\
f \circ (b\bullet) \circ (a\bullet) & \\
= & \{\ \text{defn of } (\odot)\ \} \\
a \odot (f \circ (b\bullet)) & \\
= & \{\ \text{defn of } (\odot)\ \} \\
a \odot (b \odot f) \quad \square &
\end{array}$$

Intuitively, the reason $m$ needs to be commutative is that in acting on a function $f$, it acts *contravariantly* on $f$'s arguments, so the "order of application" is switched. More precisely, lifting turns a *right action* of $m$ on $a$ into a *left action* of $m$ on $(\to)\ a$ and vice versa; for commutative monoids, left and right actions are the same.

The lifted action defined above is by no means the only way to lift an action to function types. For instance, when the monoid is a *group*, an equally natural action is given by the equation $(m \odot f)\ x = f\ (m^{-1} \bullet x)$. However, this is a very different action and is not useful for us.

Lifting monoid actions can also be generalised to arbitrary *arrows* (Hughes 1998); this generalisation will come in useful later. Given the standard definition

```
newtype WrappedArrow (⇝) a b = WrapArrow (a ⇝ b)
```

we can define a *Monoidal* instance and the generalised lifted action as follows:

```
instance Arrow (⇝)
        ⇒ Monoidal (WrappedArrow (⇝) a) where
  unit = WrapArrow $ arr (const ())
  WrapArrow f ⋆ WrapArrow g = WrapArrow (f △ g)
instance (Arrow (⇝), Action m a)
        ⇒ ActionF m (WrappedArrow (⇝) a) where
  m ⊙ WrapArrow f = WrapArrow $ f ⋘ arr (m•)
instance (Arrow (⇝), Action m a)
        ⇒ DistributiveF m (WrappedArrow (⇝) a)
```

Notice that $arr\ (m\bullet)$ is just the "shift by $m$" function lifted to an arrow, and $(\lll)$ is arrow composition. If we ignore the newtype wrappers, this is essentially the same as the definition of the lifted action for function spaces. In fact, given the arrow laws $arr\ id = id$ and $arr\ (f \circ g) = arr\ f \lll arr\ g$, we can simply take our proof for the action of $m$ on $(\to)\ a$ and generalise it to a proof for the above instance with *WrapArrow*, by suitably wrapping things in calls to *arr*, and wrapping and unwrapping the *WrapArrow* construnctor; we must also note that $(\triangle)$ generalises to an operation on arbitrary *Arrow* instances, which is required by the *Arrow* laws to satisfy $(f \mathbin{\triangle} g) \lll h = (f \lll h) \mathbin{\triangle} (g \lll h)$.

As a final note, we can also "demote" the *Monoidal* instance for *WrappedArrow* $(\rightsquigarrow)$ $a$ to a *Monoid* instance for *WrappedArrow* $(\rightsquigarrow)$ $a$ $()$; this will come in handy later.

```
instance Arrow (⇝) ⇒
         Monoid (WrappedArrow (⇝) a ()) where
  ε     = unit
  f ◇ g = (f ⋆ g) ⋙ unit
```

### 5.2 Twisted Functors

Given the foregoing machinery, we can now concisely define twisted functors. Given a monoid $m$ with a distributive action on a functor $f$, the *twisted functor* $f \boxtimes m$ is defined as follows:

```
data (⊠) f m a = f a :⊠ m

unTwist :: (f ⊠ m) a → (f a, m)
unTwist   (fa :⊠ m)  = (fa, m)
```

Just as with semi-direct products, a twisted functor structurally consists of an $f$ value paired with a monoidal tag. The *Functor* instance for $f \boxtimes m$ maps over the first value of the pair while leaving the tag alone:

```
instance Functor f ⇒ Functor (f ⊠ m) where
  fmap f (x :⊠ m) = (fmap f x :⊠ m)
```

In addition, if $f$ is *Monoidal* and the action of $m$ on $f$ is distributive, then $f \boxtimes m$ is *Monoidal*:

```
instance DistributiveF m f ⇒ Monoidal (f ⊠ m) where
  unit                       = unit :⊠              ε
  (f₁ :⊠ m₁) ⋆ (f₂ :⊠ m₂) = (f₁ ⋆ (m₁ ⊙ f₂)) :⊠ (m₁ ⋄ m₂)
```

Again, just as with semi-direct products, when combining two tagged values, the first tag acts on the second value before the values are combined.

Verifying that $f \boxtimes m$ satisfies the *Functor* and *Monoidal* laws is now straightforward. For the *Functor* laws, *fmap id* has no effect on $f \boxtimes m$ values since

$$fmap\ id\ (x :\boxtimes m) = fmap\ id\ x :\boxtimes m = x :\boxtimes m;$$

the other *Functor* law then follows by parametricity (Kmett 2015). The proof of the identity and associativity laws for the *Monoidal* instance is literally the same as the proof of the *Monoid* instance for semi-direct products (Section 4.2), with $\varepsilon_a$ replaced by *unit* and $(\bullet)$ replaced by $(\odot)$.

## 6. Offset Monoids and Their Action on Pointers

In this section, we describe the monoid action that is relevant for our pointer libraries, namely, the action of offsets on pointers. In the examples in Section 2, we used the type *Int* for keeping track of offsets measured in bytes. However, we want to be able to measure pointer offsets in units other than bytes. For example, in the case of memory allocation, we measure offsets in units corresponding to the alignment restrictions of the machine. To avoid errors due to confusion of units, offsets using different units should of course have different types.

We first define a new type *BYTES* that captures lengths measured in bytes:

```
newtype BYTES = BYTES Int
  deriving (Eq, Ord, Num, Enum)
```

However, as mentioned before, it is often more natural to measure length in other units, which will in general be some multiple of bytes. We allow arbitrary types to be used as length measurements, and simply require them to be instances of the type class *LengthUnit*, which allows conversion to bytes when needed.

```
class (Num u, Enum u) ⇒ LengthUnit u where
  inBytes :: u → BYTES

instance LengthUnit BYTES where
  inBytes = id
```

From now on, we say that a Haskell type $u$ is a *length unit* if it is an instance of the type class *LengthUnit*.

An example of a type safe length unit other than *BYTES* is the *ALIGN* type. It measures lengths in multiples of the word size of the machine, and is used to ensure alignment during allocation.

```
newtype ALIGN = ALIGN Int
  deriving (Eq, Ord, Num, Enum)
instance LengthUnit ALIGN where
  inBytes (ALIGN x) = BYTES $ x ∗ alignment (⊥ :: Word)
```

The *alignment* function computes the alignment size for any instance of the class *Storable*. As we will see, the above definition essentially ensures that we always use word boundaries as allocation boundaries, since we only shift pointers by multiplies of the alignment size for *Word*.

Now consider any length unit $u$ which in particular is a numeric type. Recall that the type *Sum u* captures the underlying additive monoid of this numeric type. This monoid acts on pointers by shifting them:

```
instance LengthUnit u ⇒ Action (Sum u) Pointer where
  a • ptr = ptr ‘plusPtr‘ offset
    where BYTES offset = inBytes $ getSum a
```

This action also lifts to functions (or arrows) taking pointers as arguments, and forms the core action used in abstracting pointer manipulations.

## 7. Revisiting Serialisation and Deserialisation

In Section 2, we considered monoids relating to the problems of byte-oriented serialization and deserialisation. We now revisit these examples and show how they fit into the framework of semi-direct products and twisted functors, as well as filling in some concrete implementation details.

Recall that the basic interface for serialisation and deserialisation is formed by *IO* actions which depend on a "current" memory location, that is, functions of type (*Pointer* → *IO a*). These can be captured by what are known as *Kleisli arrows* where the underlying monad is *IO*.

```
newtype Kleisli m a b = Kleisli {runKleisli :: a → m b}
```

A *pointer arrow* of type *PointerArr* is then essentially a function of type *Pointer* → *IO a*—though slightly obscured by the *Kleisli* and *WrappedArrow* newtypes, which allow us to reuse appropriate instances. *WriteAction* (of kind $*$) and *ParseAction* (of kind $* \to *$) can then be defined as pointer arrows returning the unit type and an arbitrary parameter type, respectively.

```
type PointerArr   = WrappedArrow (Kleisli IO) Pointer
type WriteAction = PointerArr ()
type ParseAction = PointerArr
```

(It might be clearer to define *ParseAction a* = *PointerArr a*, but type synonyms must be fully applied, and we will need to use *ParseAction* :: $* \to *$ on its own, without a type argument.) Recall that length units act on *Pointer* by shifting, as described in the previous section. The lifted action of length units on (→) *Pointer* (or *WrappedArrow* (⇝) *Pointer*), therefore, does exactly what we described intuitively in Section 2, that is, $u \odot f = f \circ (u\bullet)$ is the function which first shifts the input *Pointer* by $u$ before running the action $f$. We can therefore define our serialisation and deserialisation types as a semi-direct product and twisted functor, respectively.

```
type Write  = WriteAction ⋊ Sum BYTES
type Parser = ParseAction ⊠ Sum BYTES
```

Recall that as a wrapped arrow type returning (), *WriteAction* has a "demoted" *Monoid* instance, whereas *ParseAction* is *Monoidal*.

An appropriate *Monoid* instance for *Write* and *Monoidal* (or *Applicative*) instance for *Parse* then follow from our theory developed in Section 5. Indeed, the instances come for free, without having to write any additional code.

To make use of our data serialisation and deserialisation framework, all we need are some explicitly length tagged write and parse actions. For example, we can define write and parse actions for any *Storable* instance as follows:

```
byteSize :: Storable a ⇒ a → Sum BYTES
byteSize = Sum ∘ BYTES ∘ sizeOf

writeStorable :: Storable a ⇒ a → Write
writeStorable a = action :⋊ byteSize a
  where
    action = WrapArrow ∘ Kleisli $ pokeIt
    pokeIt = flip poke a ∘ castPtr

parseStorable :: ∀a.Storable a ⇒ Parser a
parseStorable = pa
  where
    action = WrapArrow ∘ Kleisli $ (peek ∘ castPtr)
    pa     = action :⊠ byteSize (⊥ :: a)
```

The *peek* and *poke* functions are exposed by the *Storable* type class. More complicated write and parse actions can then be built using the *Applicative* and *Monoid* interfaces.

## 7.1 Bounds Checking

Since write and parse actions always come tagged with their length, we can actually compute the number of bytes written or read by such an action as a *pure* function:

```
writeLength :: Write → BYTES
writeLength = getSum ∘ snd ∘ unSemi

parseLength :: Parser a → BYTES
parseLength = getSum ∘ snd ∘ unTwist
```

This is useful for building safe interfaces which guarantee the absence of buffer overflows. Consider the low-level *create* function provided by the `bytestring` library:

```
create :: Int                       -- size
       → (Ptr Word8 → IO ())        -- filling action
       → IO ByteString
```

This function provides a way to initialize a *ByteString* by directly accessing a pointer to the beginning of the allocated buffer, performing an arbitrary *IO* action to populate the buffer. Obviously this function provides no particular safety guarantees. However, we can wrap *create* in a safe interface by leveraging the known size of a *Write* action:

```
toByteString :: Write → IO ByteString
toByteString w      = create len fillIt
  where BYTES len = writeLength w
        fillIt      = unsafeWrite w ∘ castPtr

unsafeWrite :: Write → Pointer → IO ()
unsafeWrite = runKleisli ∘ unwrapArrow ∘ fst ∘ unSemi
```

Buffer overflows are avoided by hiding *unsafeWrite* and providing only the high-level *toByteString* to the user. Similarly, we can define a safe version of parsing, with a type like

```
parseByteString :: Parser a → ByteString → IO (Maybe a)
```

## 7.2 Strengths and Limitations

Our simple interface for data serialisation and deserialisation has several advantages.

- There is no explicit pointer arithmetic other than in the instance for the action of *Sum u* on *Pointer*.
- Once low-level combinators like *writeStorable* are defined, any compound serialisation or deserialisation combinator can be built using the *Applicative* interface, without worrying about pointer arithmetic or bounds calculations.

On the other hand, the interface does have some nontrivial limitations, and is certainly not intended as a general-purpose serialisation and parsing framework.

Haskell has high-performance serialisation libraries such as `blaze-builder` (Van der Jeugt et al.) which allows serialising data incrementally as a lazy byte string. A lazy byte string is a lazy list of chunks each of which is a strict byte string. Typically the generated lazy byte strings are written to a file or sent over a socket. However, it is not desirable to generate a large list of small chunks each of which will incur various overhead costs due to cache misses or system calls. On the other hand, using a single large chunk defeats the purpose of incremental generation. The `blaze-builder` library achieves high performance by making sure that the chunks are of reasonable size, typically the size of the L1 cache of the machine. It performs incremental writes to buffers and spills them when their size is just right.

The serialisation framework that we developed here is not directly suitable for such an application. Nonetheless, we believe that with some additional data structures, a `blaze-builder`-like interface can be built on top. Instead of keeping partially filled buffers, we can keep an element of the *Write* monoid to which we append subsequent writes via the underlying monoid multiplication. When the write we are accumulating is large enough, we generate a chunk by using the *toByteString* function. We have not explored this idea but believe such an interface could be competitive.

The limitations of our parsers, on the other hand, are much more serious. In particular, our interface *cannot* replace an incremental parsing interface like `attoparsec` (O'Sullivan). By construction, our parsers must compute up front—that is, *without* executing any parse actions—the total amount of data that will be read when the action is executed. This is what makes the *parseLength* function pure. This is a serious limitation that precludes *Alternative* or *Monad* instances for our parsers (instances of either class must be able to make choices based on intermediate results). For example, consider the following encoding of *Maybe Word*: A value of *Nothing* is encoded as a single word which is 0, and *Just w* is encoded as two words where the first word is 1 and the second word is *w*. It is rather easy to write a parser for such a format using the *Alternative* or *Monad* interfaces, but it is not possible in our framework.

Despite these limitations, many use cases that we encounter in our cryptographic library can be easily be dealt with, since most types of interest to us in the cryptographic setting are essentially product types. For example, *Sha1* is a 5-tuple of *Word32*'s. Writing parsers for such types is straightforward using the *Applicative* interface. It is also noteworthy that these very limitations also ensure that deserialisation can be automatically parallelised, since actions cannot depend on the results of previous actions, and the number of bytes needed for each deserialisation subtask is known.

One can also easily parameterise parsers on the amount they should consume, as in this example:

```
replicateA :: Applicative f ⇒ Int → f a → f [a]
replicateA n f
  | n ⩽ 0      = pure []
  | otherwise = (:) ⟨$⟩ f ⟨∗⟩ replicateA (n − 1) f

listOf :: Int → Parser a → Parser [a]
listOf = replicateA
```

Such parsers can be used to parse types like *Integer* in certain contexts, for example, when it is known that the integer is exacty 1024 bits long (think of parsing a 1024-bit RSA key).

## 8. An Interface for Secure Memory Allocation

Sensitive information like long term private keys can leak into external memory devices when the operating system swaps the pages which contain such data. Data that are written to such external media can survive for a long time. Therefore, sensitive information should be prevented from being swapped out of main memory.

Most operating systems allow a user level process to *lock* certain parts of its memory from being swapped out. To protect sensitive data from hitting long term memory, a cryptographic library can lock the memory where it stores such data. When this data is no longer needed, the application should wipe the memory clean before unlocking and de-allocating it. Typical operating systems usually place stringent limits on the amount of memory that a user level process can lock, in order to prevent users from abusing such provisions to consume system resources at the cost of others. Thus locked memory must be treated as a precious resource and carefully managed.

A naive way to perform memory locking in Haskell is to use the pointer type *ForeignPtr*, which allows one to assign customised finalisation routines. For a *ForeignPtr* that stores sensitive data, one needs to lock it before use and associate a finalisation routine that wipes the memory and unlocks it. However, such a naive implementation does not work. In typical operating systems, locking and unlocking happens at the virtual memory management level, with the following consequences:

1. Memory locking is possible only at the *page level* and not at the byte level.
2. Calls to locking and unlocking usually do not nest. For example, in a POSIX system, a single *munlock* call on a page is enough to unlock multiple calls of *mlock* on that page.

Now consider two distinct memory buffers $b_1$ and $b_2$ that contain sensitive data. Although $b_1$ and $b_2$ do not overlap in memory, they can often share a page. For example, $b_1$ might end at the first byte of a page and $b_2$ might be the rest of the page. In such a situations naively unlocking the memory referenced by $b_1$ when it is finalised will unlock $b_2$, even though $b_2$ is still in use and hence not ready to be unlocked. Clearly this is undesirable and hence such a simple finalisation-based solution is not enough to secure sensitive information from being swapped out.

To avoid these problems, we could write a small memory management subsystem within our library that allocates locked pages of memory from the operating system and distributes them among the foreign pointers that are used to store sensitive data. As these foreign pointers are finalised, the library should mark the appropriate memory locations as free and perform garbage collection and compaction. However, such a solution is difficult to maintain and requires knowing system parameters such as page sizes and page boundaries, which we would like to avoid.

In this section, we look at a simpler approach to securing memory which does not suffer from these complications. The important components of our memory subsystem are as follows.

**Memory elements:** A *memory element* is an abstract type containing a memory buffer. Memory elements are instances of the type class *Memory* (to be described shortly). Simple memory elements, like the type *MemoryCell a* which is capable of storing a single value of type *a*, are provided by the library. In general, however, a memory element could be a product of multiple such primitive memory elements.

In our library, operations that need secure memory are encoded as functions of type $mem \to IO\ a$ for some appropriate memory element *mem*. All basic cryptographic operations such as hashing or encryption have their own memory elements. Since memory elements can be composed, a cryptographic operation that needs "multiple memory elements" can just take as input a single compound memory element (just as multi-argument functions can be encoded as single-argument functions expecting a tuple).

**Allocation strategy:** Each memory element has an associated allocation strategy, which governs how much memory is needed and how it is to be allocated among its sub-elements. For a memory element *mem*, the allocation strategy, captured by the type *Alloc mem*, turns out to be a twisted functor. This applicative interface comes in handy when we design allocation strategies for compound memory elements: if $mem_1$ and $mem_2$ are two memory elements with allocation strategies $a_1 :: Alloc\ mem_1$ and $a_2 :: Alloc\ mem_2$ respectively, the allocation strategy for the product type $(mem_1, mem_2)$ is simply given by $a_1 \star a_2$.

The *Memory* class by itself is very simple; instances of *Memory* are types with an allocation strategy and a way to recover the underlying pointer.

```
class Memory mem where
  memoryAlloc  :: Alloc mem          -- allocation strategy
  underlyingPtr :: mem → Pointer     -- recover the pointer
```

We now define the type *Alloc mem*, which represents an allocation strategy for the memory element *mem*. We begin by defining an allocation action which essentially is a function that takes takes a pointer to a block of memory and carves out an element of type *mem*.

```
type AllocAction = WrappedArrow (→) Pointer
```

Note *AllocAction* itself does not involve *IO*. An *AllocAction* does not actually allocate memory via the operating system, but simply builds an appropriate value (such as a buffer, memory cell, *etc.*) given a pointer to an already-allocated block of memory.

However, the induced *Applicative* instance for *AllocAction* is insufficient. We also need to keep track of the total memory allocated so we do not allocate the same memory twice: when allocating two memory elements, we can shift the pointer by the length of the first element before allocating the second. As in the previous examples, we can achieve this interface with a twisted functor.

```
type Alloc = AllocAction ⊠ Sum ALIGN

allocSize :: Alloc m → BYTES
allocSize = inBytes ∘ getSum ∘ snd ∘ unTwist
```

The only difference from *Parser* is that we measure offsets in terms of the alignment boundaries of the architecture and not bytes. Hence, we have used *Sum ALIGN* as the monoid instead of *Sum BYTES*.

We now look at how actual memory allocations are done. Recall that generic cryptographic operations are represented as functions of type $mem \to IO\ a$, where *mem* is an instance of *Memory*. Thus, *mem* comes equipped with an allocation strategy *memoryAlloc* :: *Alloc mem* which, in particular, tells us the size of the buffer that must be allocated for the memory element. When given a block of memory of the appropriate size, the allocation strategy also knows how to wrap it to create the actual memory element. We can thus define a high level combinator *withSecureMemory* which performs memory allocation and then runs an action:

```
withSecureMemory :: Memory mem
                 ⇒ (mem → IO a) → IO a
```

The use of the *withSecureMemory* combinator follows a standard design pattern in Haskell for dealing with resource allocation: one first builds up an *IO* action which takes a resource as argument— in this case the memory element— and then dispatches it using *withSecureMemory* at the top level. *withSecureMemory* then allocates and locks the proper amount of memory, turns it into a *mem* value, and passes it to the provided function. After the *IO* action has finished executing, it makes sure that the allocated memory is wiped clean—even if the *IO* action fails with an exception—and then unlocks and deallocates it. We skip the details of the implementation, as it involves low level memory allocation and locking which is not very enlightening.

The interface we describe solves the problems we outlined before provided one sticks to the idiom of using *withSecureMemory* only at the top level; there can only ever be one active call to *withSecureMemory*, and hence there can never be overlapping or nested locks and unlocks. If an algorithm has multiple disjoint phases requiring secure memory, the implementor can choose to implement the algorithm with multiple sequential calls to *withSecureMemory*. However, one should not nest calls of *withSecureMemory*. For an algorithm that requires several overlapping uses of secure memory, it is up to the implementor to consolidate the necessary secure memory elements into a single data structure which will be allocated by a single enclosing call to *withSecureMemory*. Although this sounds tedious, in practice it is straightforward due to the *Applicative* interface of *Alloc*. The implementor simply needs to combine component allocators together via the *Applicative* API, automatically resulting in an allocator which knows the total size of the required secure memory block and knows how to lay out the components within the allocated block.

There is only one small problem—the type system does not rule out nested calls to *withSecureMemory*, which could destroy the API's guarantees (due to the problems with nested locks/unlocks, as explained previously). Since we are explicitly *not* guarding against malicious *implementors*, this is not inherently a security flaw, but one might still worry about accidentally nesting calls to *withSecureMemory*, especially as the code becomes more complex. To rule this out, one could replace the inner *IO* with another, more restricted monad which only allowed, say, doing certain pointer manipulations.

```
withSecureMemory :: Memory mem
                 ⇒ (mem → PointerIO a) → IO a
```

Critically, *PointerIO* should disallow running arbitrary *IO* actions, and in particular it should disallow nested calls to *withSecureMemory*. The current version of the library has not implemented such sanitised pointer manipulation monad but we may explore it in future versions of the library.

We now give some examples to demonstrate how our interface simplifies the definition of memory objects. Consider the following primitive memory element type exposed by our library.

```
newtype MemoryCell a = MemoryCell {unCell :: Pointer}
```

For a type *a* that is an instance of *Storable*, *MemoryCell a* is an instance of *Memory*, with the code shown below. Essentially, it just wraps a *Pointer* to a block of allocated secure memory in a *MemoryCell*, with a bit of extra complication due to the need to compute a *Word*-aligned size big enough to hold the value of type *a*:

```
instance Storable a ⇒ Memory (MemoryCell a) where
  memoryAlloc  = makeCell
  underlyingPtr = unCell
makeCell :: ∀b.Storable b ⇒ Alloc (MemoryCell b)
makeCell = WrapArrow MemoryCell
          :⊠ atLeast (sizeOf (⊥ :: b))
atLeast :: Int → Sum ALIGN
atLeast x
  | r > 0      = Sum ∘ ALIGN $ q + 1
  | otherwise  = Sum ∘ ALIGN $ q
  where (q, r) = x ‘quotRem‘ alignment (⊥ :: Word)
```

This primitive definition of *MemoryCell* involves some fiddly explicit bounds tagging. However, other memory elements that require multiple such memory cells need not do all these bound calculations, thanks to the *Applicative* instance of *Alloc*.

For example, consider the memory element that is required to compute the SHA1 hash of a streaming input. The hashing algorithm considers the data as a stream of blocks of 64 bytes each. It starts with a default value for the hash, repeatedly reading in a new block of data and computing the new hash from the hash of the blocks seen so far. The memory element that we need for such a computation involves memory cells to keep track of (1) the hash of the blocks seen so far and (2) the total number of bytes processed, which is used at the end for padding. We capture this using the following memory type.

```
data SHA1
  = SHA1 Word32 Word32 Word32 Word32 Word32
  -- Storable SHA1 instance skipped
data Sha1Mem = Sha1Mem
  {sha1Cell   :: MemoryCell SHA1
  , lengthCell :: MemoryCell Word64
  }
```

The applicative interface for *Alloc* gives us a simple way to define the *Memory* instance for *Sha1Mem* that does not require any pointer arithmetic or bound calculations, since the instance itself takes care of all the necessary bookkeeping.

```
instance Memory Sha1Mem where
  memoryAlloc  = Sha1Mem ⟨$⟩ memoryAlloc
                         ⟨∗⟩ memoryAlloc
  underlyingPtr = underlyingPtr ∘ sha1Cell
```

## 9. A Categorical Perspective on the Twisted Functor Laws

For those readers familiar with category theory, we close by briefly explaining the categorical viewpoint that motivates the particular laws we have chosen.

Consider the action of a monoid $M$ on another monoid $N$. The distributivity law is required to show that the semi-direct product $N \rtimes M$ is indeed a monoid. It is well known that there is a category theoretic interpretation for this law. In this section, we show that both the uniformity law required for monoid acting on a functor and the distributivity law for applicative functor have category theoretic motivations as well.

For a monoid $M$, by defining an action on $A$ our aim is to think of elements of $m$ as transformations of $A$. If $A$ belongs to a certain category $\mathcal{C}$, we need to associate elements $M$ to endomorphisms of $A$ in a natural way. In other words, an action of $M$ on $A$ is a homomorphism from $M$ to the monoid $\mathsf{Hom}_{\mathcal{C}}(A, A)$ of endomorphisms of $A$. Equivalently, for each element $m$ in $M$, we need to assign an endomorphism $\widehat{m}$ in $\mathsf{Hom}(A, A)$ such that $\widehat{m_1 . m_2} = \widehat{m_1} \circ \widehat{m_2}$. When $\mathcal{C}$ is the category of sets all we need is therefore the triviality and composition laws mentioned Section 5. However, when we look at objects with additional structures, we need additional constraints on $\widehat{m}$. For example, when we consider the monoid $M$ acting on a monoid $A$, then each $m$ in $M$ should define a homomorphism from $A$ to itself. This condition is captured by the identity and distributivity law for monoid acting on monoids.

It turns out that the uniformity laws that we had for monoid actions on functors as well as the additional distributivity constraint for applicative functors also have similar motivations. For two categories $\mathcal{C}$ and $\mathcal{D}$, consider the category of functors from $\mathcal{C}$ to $\mathcal{D}$ with $\mathsf{Hom}(F, G)$ being natural transformations from the functor $F$ to the functor $G$. Therefore, a monoid action of $M$ on a functor $F$

should assign, for each element $m$ of $M$, a natural transformation $\widehat{m}$ from the functor $F$ to itself. As a result, for any two objects $A$ and $B$ of $\mathcal{C}$ and any morphism $f$ from $A$ to $B$, $\widehat{m} \circ F(f) = F(f) \circ \widehat{m}$ as morphisms in $\mathsf{Hom}_{\mathcal{D}}(F(A), F(B))$. It is this condition that becomes the uniformity law.

## 10. Related Work

Monoids are mathematical constructs that have seen a wide range of applications in theory (for example, see Pin (1997) for connections to automata theory and regular languages) and practice (for example, Hinze and Paterson (2006)). Haskell libraries are full of *Monoid* instances. In particular, using *Monoid* as an interface for data serialisation is itself well known in the Haskell world, with two high performance libraries—namely, `blaze-builder` (Van der Jeugt et al.) and `binary` (Kolmodin) using monoids as their primary interface. Monoid actions have been used extensively in the design of the `diagrams` library, a DSL for describing vector graphics in Haskell. The functional pearl by Yorgey (2012) discusses this and other applications of monoids and monoid actions.

The semi-direct product is a well known construction in group theory and occurs naturally in many contexts. For example, the dihedral group $D_n$, which encodes the symmetries of a regular $n$-gon, is a semi-direct product. It is therefore natural to look for generalisation in the setting of monoidal categories. Instead of a monoid $m$ acting on another monoid $n$, for twisted functors we generalised $n$ to a monoidal functor. Other generalisations are also possible. We particularly note a generalisation studied by Paterson (2012, Section 4). This construction starts with a monoidal functor $F$ and a parameterised monoid $G$, that is, a functor $G$ that comes with a monoid multiplication $G a \times G a \rightarrow G a$, with $F$ acting on $G$. He then obtains an applicative structure on the product type $F a \times G a$. Although superficially similar to our construction—Paterson explicitly claims that his construction is a generalisation of semi-direct products—the final structure obtained is quite different, and the exact relation with our construction is unclear. In particular, given a monoid $m$ acting on a monoid $a$, we generalise $a$ to a monoidal functor, whereas Paterson generalises $m$ to a monoidal functor and $a$ to a parameterised monoid.

## 11. Conclusions and Future Work

As a running example, we concentrated on the parts of the `raaz` cryptographic library that deal with pointer arithmetic. Refactoring all the pointer arithmetic into the few lines that implement the monoid action gives us confidence that we have preempted a large class of potential memory bugs in the resulting library. It illustrates the power and flexibility of using a few core concepts like monoids and applicative functors to structure code.

There is also an indirect benefit of our approach: the fact that the type *Write* satisfies the monoid laws is simply a corollary of the fact that semi-direct products are monoids. Similarly, the functors *Parser* and *Alloc* satisfy applicative functors laws because they are twisted functors. In general, in Haskell there is no way to ensure that a purported instance of a type class indeed satisfies the associated laws. These are merely *social contracts* that any law-abiding library writer is supposed to honour to save the users of her library from mysterious bugs. The users expect the library writer to verify the laws at least informally using paper and pencil. Constructing instance compositionally as in our approach saves library writers from having to do such special case verification.

The wide applicability of monoid actions and twisted functors in capturing (almost) all pointer manipulation in our library makes us speculate that the constructions described here can be of independent interest in other programming situations. It looks like this abstraction can be used in any application that needs to keep track of some sort of location in an abstract space; the example with diagrams *Trail*s hints at this sort of generality. We would like to explore other applications of semi-direct products and twisted functors (for example, we suspect that there are much deeper applications lurking within the `diagrams` code base).

The fact that our buffer-oriented parsing framework must know the exact number of bytes to be read up front is a major limitation. Motivated by this example, we would like to explore whether the twisted functor construction could somehow be extended to *Alternative* in an appropriate way, in order to allow failure and choice.

## Acknowledgments

We thank the anonymous reviewers for their helpful comments and suggestions.

## References

S. Chauhan and P. P. Kurur. The raaz cryptographic library for Haskell. `https://hackage.haskell.org/package/raaz`.

R. Hinze and R. Paterson. Finger trees: A simple general-purpose data structure. *J. Funct. Program.*, 16(2):197–217, Mar. 2006. ISSN 0956-7968. doi: 10.1017/S0956796805005769. URL `http://dx.doi.org/10.1017/S0956796805005769`.

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

E. Kmett. The free theorem for fmap. `https://www.schoolofhaskell.com/user/edwardk/snippets/fmap`, February 2015.

L. Kolmodin. The binary library for Haskell. `https://hackage.haskell.org/package/binary`.

C. McBride and R. Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, Jan. 2008. ISSN 0956-7968. doi: 10.1017/S0956796807006326. URL `http://dx.doi.org/10.1017/S0956796807006326`.

B. O'Sullivan. The attoparsec library for Haskell. `https://hackage.haskell.org/package/attoparsec`.

R. Paterson. *Constructing Applicative Functors*, pages 300–323. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. URL `http://dx.doi.org/10.1007/978-3-642-31113-0_15`.

J.-E. Pin. Syntactic semigroups. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages, Vol. 1*, pages 679–746. Springer-Verlag New York, Inc., New York, NY, USA, 1997. ISBN 3-540-60420-0. URL `http://dl.acm.org/citation.cfm?id=267846.267856`.

S. D. Swierstra and L. Duponcheel. Deterministic, error-correcting combinator parsers. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Advanced Functional Programming*, volume 1129 of *LNCS-Tutorial*, pages 184–207. Springer-Verlag, 1996.

J. Van der Jeugt, S. Meier, and L. P. Smith. The blaze builder library for Haskell. `https://hackage.haskell.org/package/blaze-builder`.

P. Wadler. Theorems for free! In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 347–359. ACM, 1989.

R. Yates and B. A. Yorgey. Diagrams: a functional EDSL for vector graphics. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Functional Art, Music, Modelling and Design*, pages 4–5. ACM, 2015.

B. Yorgey. The monoid-extras library for Haskell. `https://hackage.haskell.org/package/monoid-extras`.

B. A. Yorgey. Monoids: Theme and variations (functional pearl). In *Proceedings of the 2012 Haskell Symposium*, Haskell '12, pages 105–116, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1574-6. doi: 10.1145/2364506.2364520. URL `http://doi.acm.org/10.1145/2364506.2364520`.