Applicative Data-Driven Computation

Conal Elliott
LambdaPix
conal@conal.net

Abstract

Graphical user interfaces (GUIs) are usually programmed in an "unnatural" style, in that implementation dependencies are inverted, relative to logical dependencies. We suggest that this reversal results directly from the imperative, data-driven orientation of most GUI libraries. While outputs depend on inputs from a user and semantic point of view, the data-driven approach imposes an implementation dependence of inputs on outputs.

This paper presents simple, functional interfaces for data-driven programming in general and GUI programming in particular, in which program dependencies directly mirror logical dependencies. The interfaces are structured as *applicative functors (AFs)*, rather than monads or arrows. Efficiency is retained while abstracting the mechanics of data-driven computation out of client programs and into reusable library code. The implementations of data-driven computation and of GUIs are also quite simple, largely due to structuring them as *compositions* of AFs.

This paper is in draft stage. I'd love to get your comments, especially via the paper's wiki "talk page", where you can find other comments as well.[1]

1. Simple data-driven computation

Imperative programs implement data-driven computation using two mechanisms: value extraction and change notification. Value extraction allows retrieval of a "current value" (e.g., via an input widget's access method). Notification allows various states (e.g., an output widget) to be updated, making them consistent with newly changed values. Our representation of data-driven computations encapsulates these two mechanisms, building them in tandem using a familiar set of combinators.

1.1 Extractors

Value extractors is represented simply as IO values.

```
type Extractor = IO
```

For example, given a reference $r :: IORef\ Int$, define the extractor $rx = readIORef\ \ r$. Or, given a slider widget s, define the extractor $sx = get\ selection\ s$.[2]

We can combine extractors applicatively. For instance, the following function defines a "sum" of extractors, i.e., an extractor whose current value is the sum of the current values of given ones.

```
plusX :: Num a ⇒
   Extractor a → Extractor a → Extractor a
plusX rx sx = do r ← rx
                 s ← sx
                 return (r + s)
```

This code is quite tedious to write, so we would prefer to use the $liftM_2$ higher-order function, defined for monads:

```
plusX = liftM₂ (+)
```

Instead of this monad-based formulation, we use a more general formulation in terms of "applicative functors" (AFs) [2]. [3] The AF formulation of $plusX$ looks much like the monadic formulation:

```
plusX = liftA₂ (+)
```

It's also easy to wrap up a regular value as an extractor. Formulated monadically, we'd simply use $return$. The more general AF formulation is "$pure$". Thus, using AF methods, one can write arbitrarily rich applicative expressions to denote extractors.[4]

1.2 Notifiers

For efficient data-driven computation, value extraction is not enough; we also need to construct change notifiers. We will represent a notifier as the ability for clients to "subscribe" actions to be invoked whenever an event occurs.

```
type Notifier = IO () → IO ()
```

The following function is handy for creating nontrivial sources. It makes a notifier, given a "$setNotify$" function that (destructively) assigns a single action to be executed upon some event. The subscribing actions are accumulated into a single, sequenced action held in a reference.[5]

```
mkNotifier :: Notifier → IO Notifier
mkNotifier setNotify =
  do ref ← newIORef (return ())   -- subscribed actions
     setNotify (join (readIORef ref))
     return $ modifyIORef ref ∘ (≫)
```

---

[1] `http://haskell.org/haskellwiki/Talk:Applicative_data-driven_programming`

[2] The low-level GUI mechanisms are handled by wxHaskell [1].

[3] The $Applicative$ interface has just two operations: injection of a pure value and a form of function application.

```
class Functor f ⇒ Applicative f where
  pure  :: a → f a
  (<*>) :: f (a → b) → f a → f b
```

These primitives are used to define generalizations of the monadic $liftM$, $liftM_2$, etc.

```
liftA f a       = pure f <*> a
liftA₂ f a b    = liftA f a <*> b
liftA₃ f a b c  = liftA₂ f a b <*> c
...
```

[4] *[Consider adopting the AF sugar for this paper.]*

[5] Note that $join :: Monad\ m \Rightarrow m\ (m\ a) \rightarrow m\ a$, so $join$ here turns an $IO\ (IO\ ())$ into an $IO\ ()$ that both reads the reference *and* executes the contained value. The last line returns an action that modifies the contents of the reference by sequencing its current action with a new one.

For example, imperative GUI toolkits come with a way to specify a "callback" action to invoke when a widget is modified. Providing the widget and abstracting over the action gives a *setNotify* function suitable for passing to *mkNotifier*.

```
cmdNotifier :: Commanding wid ⇒ wid → IO Notifier
cmdNotifier wid =
  mkNotifier (λact → set wid [on command := act])
```

Given atomic notifiers (e.g., as constructed from a widget and *mkNotifier*), how do we build notifiers compositionally? From notifiers *rn* and *sn*, we'd like to construct a composite notifier that reports a change whenever *rn or sn* reports a change. Longhand,

```
orN :: Notifier → Notifier → Notifier
orN rn sn = λact → rn act ≫ sn act
```

We'll also want to make a notifier for never-occurring events, such as a pure (immutable) value changing. Longhand,

```
neverN :: Notifier
neverN = λact → return ()
```

Just as with extractors, we prefer to compose notifiers in terms of a more generic interface. Instead of *Monad* or *Applicative*, we use *Monoid*.

Exploiting the *Monoid* instances for functions, *IO a*, and (), we have the following simple definitions.[6]

```
neverN = mempty
orN    = mappend
```

We now abandon the names "*neverN*" and "*orN*", and simply use "*mempty*" and "*mappend*".

1.3 Combining the pieces

Our representation of data-driven computations pairs the representations given above for extractors and notifiers, into a "source" of values. A single set of combinators works on both representations in tandem. For reasons explained below, we will place the notifier first and apply a **newtype** constructor "*O*" to the pair. For instance, a sum of two sources:

```
addS :: Num a ⇒ Source a → Source a → Source a
addS (O (rn, rx)) (O (sn, sx)) =
  O (rn ‘mappend‘ sn, liftA₂ (+) rx sx)
```

To make a *source* from a value (unchanging) *v*,

```
pureS :: a → Source a
pureS a = O (mempty, pure a)
```

There is, again, a much more succinct formulation, made possible by casting *Source* as another AF.

```
addS  = liftA₂ (+)
pureS = pure
```

The key to these simple definitions is to define *Source* as a type composition:

```
type Source = (, ) Notifier ◦ Extractor
```

where type composition is defined as follows.

```
newtype (g ◦ f) a = O{ unO :: g (f a)}
```

Using a **newtype** rather than a type synonym enables exploiting some general properties of type composition. In particular, compositions of functors are functors, and compositions of AFs are AFs [2, Section 5].

```
instance (Functor g, Functor f) ⇒ Functor (g ◦ f) where
  fmap h (O gf) = O (fmap (fmap h) gf)
instance (Applicative g, Applicative f)
   ⇒ Applicative (g ◦ f) where
  pure a              = O (pure (pure a))
  O getH <∗> O getX = O (liftA₂ (<∗>) getH getX)
```

Sometimes we'll want to apply a function *h* under the *O* constructor:

```
inO :: (g (f a) → g′ (f′ a′)) → (O g f a → O g′ f′ a′)
inO h = O ◦ h ◦ unO
```

These composition properties are applicable because pairing with *Notifier* is an AF, which is the case exactly because *Notifier* is a monoid.

```
instance Functor ((, ) u) where
  fmap f (u, x) = (u, f x)
instance Monoid u ⇒ Applicative ((, ) u) where
  pure x              = (mempty, x)
  (u, f) <∗> (v, x) = (u ‘mappend‘ v, f x)
```

By combining the instances for *g*◦*f* with the instances for (,) *u* specialized to *Notifier*, it follows that, for sources,

```
fmap f (O (rn, rx)) ≡ O (rn, fmap f rx)
pure a ≡ O (mempty, pure a)
O (nf, xf) <∗> O (nx, xz) ≡
  O (nf ‘mappend‘ nx) (xf <∗> xz)
```

Returning to the sum example above, the previous definitions of *addS* and *pureS* can now be derived.

```
  pureS a
≡ pure a
≡ O (pure (pure a))
≡ O (mempty, pure a)
```

and

```
  addS (O (rn, rx)) (O (sn, sx))
≡ liftA₂ (+) (O (rn, rx)) (O (sn, sx))
≡ fmap (+) (O (rn, rx)) <∗> O (sn, sx)
≡ O (fmap (fmap (+)) (rn, rx)) <∗> O (sn, sx)
≡ O (rn, fmap (+) rx) <∗> O (sn, sx)
≡ O (liftA₂ (<∗>) (rn, fmap (+) rx) (sn, sx))
≡ O (rn ‘mappend‘ sn, fmap (+) rx <∗> sx)
≡ O (rn ‘mappend‘ sn, liftA₂ (+) rx sx)
```

Beside *pure* and (<∗>), we can also construct sources explicitly. For example, the following function presents a widget and input attribute as a source.

```
attrSource :: Commanding wid ⇒
  wid → Attr wid a → IO (Source a)
```

---

[6] The instances:

```
instance Monoid b ⇒ Monoid (a → b) where
  mempty        = const mempty
  f ‘mappend‘ g = λx → f x ‘mappend‘ g x

instance Monoid a ⇒ Monoid (IO a) where
  mempty  = pure mempty
  mappend = liftA₂ mappend

instance Monoid () where
  mempty          = ()
  () ‘mappend‘ () = ()
```

Note that the *IO* instance fits a more general pattern, in which *IO* may be replaced by *any* AF. In particular, the function ($a \to b$) instance is also an example of this pattern, considering the meanings of *pure* and $liftA_2$ for functions.

```
attrSource wid attr =
  do nfy ← cmdNotifier wid
     return (O (nfy, get wid attr))
```

1.4 Generalizing

In fact, the *Functor* and *Applicative* instances for *Source* rely on very little about the choice of *IO* and *Notifier*, so they can be stated much more generally.

```
type DataDriven nfr xtr = (,) nfr ∘ xtr
type Source = DataDriven Notifier Extractor
```

With this refactoring, *DataDriven nfr xtr* is an AF for *any* monoid *nfr* and applicative functor *xtr*.

1.5 Running a data-driven computation

We can "run" a source of actions by executing its current value whenever it changes.

```
runDD :: Source (IO ()) → IO ()
runDD (O (nfr, xtr)) = nfr act ≫ act
  where act = join xtr
```

Again, *join* here turns an *IO* (*IO* ()) into *act* :: *IO* (). Executing *act* retrieves *and* executes the current value of *xtr*. The body of the definition subscribes *act* and executes it once up front, as initialization.

1.6 Unique notification

As defined above, notifiers can get invoked redundantly. Consider $a + a$, where the source $a = O\ (na, xa)$. The notifier would be *na* ‘*mappend*‘ *na*, which is equivalent to $\lambda act \rightarrow na\ \ act \gg na\ \ act$. That is, any subscribing action *act* would get invoked twice.

To eliminate redundant unification, represent notifiers as maps from unique tags to simple notifiers.

```
type UNotifier = Map Int Notifier
type USource   = DataDriven UNotifier Extractor
```

Since *Map k v* is a monoid whenever *Ord k* (e.g., $k \equiv Int$), *USource* is an AF. (The *mappend* operation for maps is a left-biased union.)

To convert from *Notifier* to *UNotifier*, make a singleton map with a given tag. Conversely, to convert from *UNotifier* to *Notifier*, just forget the tags and combine the individual notifiers, which corresponds to the *fold* operation in the *Map k v* instance of *Foldable* type class (when *v* is a monoid). Using these simple conversions, define conversions between *Source* and *USource* as follows.[7]

```
toUSource :: Int → Source a → USource a
toUSource tag = inO (first (singleton tag))

fromUSource :: USource a → Source a
fromUSource = inO (first fold)
```

We'll need a way to generate generators of new tags:

```
type NewTag = IO Int

newNewTag :: IO NewTag
newNewTag =
  do symRef ← newIORef 0
     return (do modifyIORef symRef (+1)
                readIORef symRef)
```

1.7 Revisiting extractors

We've used $Extractor \equiv IO$, but extractors only *read* state, they do not write it. As such, *fx* ⊛ *ax* is insensitive to order of extraction of *fx* vs *ax*. Is there an alternative to *IO* that captures this property?nnnnn

2. GUIs, first version

We represent GUIs as functions that take a container sub-window and produce a layout and a value source.

```
type Win = Panel ()   -- widget container
type UI′ a = Win → IO (Layout, Source a)
```

From the first definition, we can see that *UI′* is a composition of four simpler components: sources, pairing with a layout, *IO*, and function from *Win*. Writing this composition explicitly will make it easy to define UI operations.

```
type UI = (→) Win ∘ IO ∘ (,) Layout ∘ Source
```

These two types are isomorphic:

```
ui   :: UI′ a → UI  a
unUI :: UI  a → UI′ a

ui   = O    ∘ O    ∘ O
unUI = unO ∘ unO ∘ unO
```

Recall from Section 1.3 that *UI* is an AF if the composed pieces are. All four pieces are indeed AFs, assuming *Layout* is a monoid. For now we'll provide a simple *Monoid* instance for *Layout*, stacking vertically:

```
instance Monoid Layout where
  mempty  = empty
  mappend = above

above, leftOf :: Layout → Layout → Layout
la ‘above‘  lb = fill (column 0 [la, lb])
la ‘leftOf‘ lb = fill (row    0 [la, lb])
```

2.1 Widgets

Input widget construction takes an initial value and makes a UI

```
type IWidget a = a → UI a
```

For instance, a string entry widget:

```
stringEntry :: IWidget String
```

Other parameters may be necessary as well, such as the value bounds for a slider.

```
islider :: (Int, Int) → IWidget Int
```

The definitions are easy, given an auxiliary function *iwidget*.

```
stringEntry = iwidget textEntry text

islider (lo, hi) =
  iwidget (λwin → hslider win True lo hi) selection
```

Beside the initial value, the function *iwidget* takes a widget-making function and a choice of attribute. Output widgets are created similarly, and the following type definition captures the commonality.

```
type MkWidget wid a b =
  (Win → [Prop wid] → IO wid) → Attr wid a → b
```

Creation of input widgets is straightforward, using *attrSource*, from Section 1.3.

---

[7] The *first* function applies a given function to the first member of a pair: $\mathit{first}\ f\ (x, y) = (f\ x, y)$. More generally, it applies to any arrow, not just to functions.

```
iwidget :: (Commanding wid, Widget wid) ⇒
    MkWidget wid a (IWidget a)
iwidget mkWid attr initial = ui $ λwin →
  do wid ← mkWid win [attr := initial]
     src ← attrSource wid attr
     return (hwidget wid, src)
hwidget :: Widget w ⇒ w → Layout
hwidget = hfill ∘ widget
```

While input widgets *produce* values, output widgets *consume* them.

```
type OWidget a = UI (a → IO ())
owidget :: Widget wid ⇒ MkWidget wid a (OWidget a)
owidget mkWid attr = ui $ λwin →
  do wid ← mkWid win []
     return (hwidget wid
            , pure (λa → set wid [attr := a]))
```

The beauty of this definition of *OWidget* is that outputs (consumers) can simply be *applied to* inputs (producers), using the central applicative functor operator, “<*>”.

For instance, we can display a string or any showable value.

```
stringDisplay :: OWidget String
stringDisplay = owidget textEntry text
showDisplay :: Show a ⇒ OWidget a
showDisplay = fmap (∘show) stringDisplay
```

2.2 Titling

Adding a title to a GUI requires altering the layout produced. The function *onLayout*, below, applies a given function to the layout part of a UI.

```
type Unop a = a → a
onLayout :: Unop Layout → Unop (UI a)
onLayout f = ui ∘ (fmap ∘ fmap ∘ first) f ∘ unUI
```

The *fmap*s correspond to the the functors (→) *Win* and *IO*, and *first* to (,) *Layout*.[8]

Adding a title then is easy, using wxHaskell’s function *boxed* :: *String* → *Layout* → *Layout*.

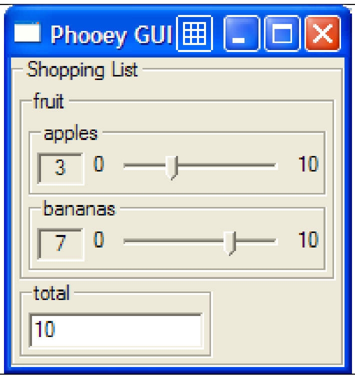
```
title :: String → Unop (UI a)
title str = onLayout (boxed str)
```

2.3 Examples

As an example, Figure 1 is a simple shopping list GUI. The total displayed at the bottom of the window always shows the sum of the values of the apples and bananas input sliders. When a user changes the inputs, the output updates accordingly.

In the code below, note that *shopping* uses the reverse application operator (<**>). This reversal causes the function to appear after (below) the argument.

```
apples, bananas, fruit :: UI Int
apples  = title "apples"  $ islider (0, 10) 3
bananas = title "bananas" $ islider (0, 10) 7
fruit   = title "fruit"   $ liftA2 (+) apples bananas
total :: Show a ⇒ OWidget a
total = title "total" showDisplay
```



apples 3

bananas 7

total 10

Figure 1. Simple GUI

```
shopping :: UI (IO ())
shopping = title "Shopping List" $ fruit <**> total
```

3. Flexible layout

So far, our generated GUIs are all laid out from top to bottom. Next we add choice of layout with the ability to mix different layouts in a GUI. The vital change is in the layout information generated for each GUI. Rather than using a fixed *Layout* monoid (*empty* and *above*), GUIs will take the monoid specification from context.

```
type UI′ a = Win → IO (CxLayout, Source a)
type UI = (→) Win ∘ IO ∘ (,) CxLayout ∘ Source
type CxLayout = CxMonoid Layout
newtype CxMonoid a =
  CxMonoid{unCxMonoid :: MonoidDict a → a}
type MonoidDict a = (a, a → a → a)
instance Monoid (CxMonoid a) where
  mempty = CxMonoid (λ(e, _) → e)
  CxMonoid f `mappend` CxMonoid g =
    CxMonoid (λmd@(_, op) → f md `op` g md)
```

The definitions of *MonoidDict* and *CxMonoid*, as well as the *Monoid* instance for *CxMonoid*, are all mechanically derived from the *Monoid* type class.

As required for *UI* to be an applicative functor, *CxLayout* is a monoid.

Running a UI works as in Section 2, except that the *MoinoidDict* (*empty*, *above*) is passed in to extract a layout.

The only change in widget creation (relative to Section 2.1) is that the new versions of *iwidget* and *owidget*, use a new function *widgetCXL* that ignores an incoming *MonoidDict*.

```
widgetCXL :: Widget w ⇒ w → CxLayout
widgetCXL wid = CxMonoid (const (hwidget wid))
```

The *iwidget* and *owidget* functions use *widgetCXL* in place of *hwidget*.

The pay-off in the new representation comes in definability of layout-altering functions. For instance,

```
leftToRight, topToBottom, flipLayout :: Unop (UI a)
leftToRight   = withCxMonoid (empty, leftOf)
topToBottom   = withCxMonoid (empty, above)
flipLayout    = compCxMonoid (second flip)
```

The *withCxMonoid* function overrides an inherited layout monoid, using the more general *compCxMonoid*.

---

[8] If we wanted to alter the value source, we would have used *second* or another *fmap* in place of *first*.

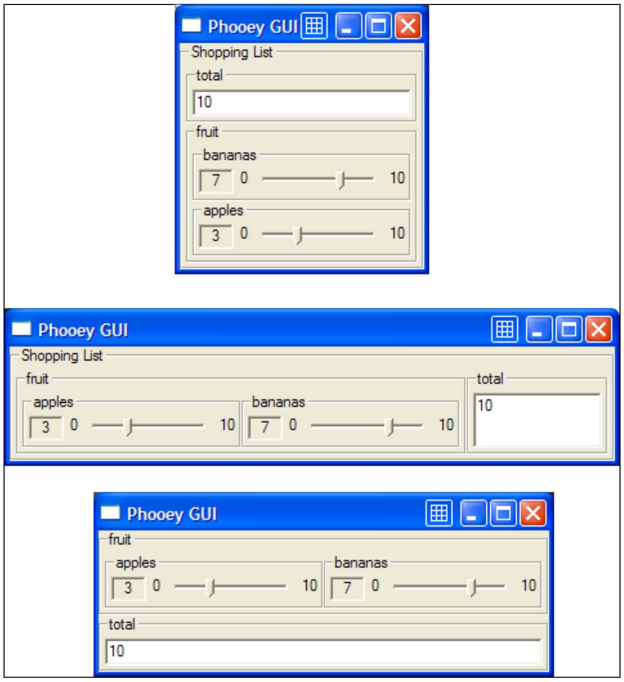


Figure 2. Some layout variants

```
withCxMonoid :: MonoidDict Layout → Unop (UI a)
withCxMonoid dict = compCxMonoid (const dict)

compCxMonoid :: Unop (MonoidDict Layout) → Unop (UI a)
compCxMonoid f = onCxLayout′ (∘f)
```

The *onCxLayout′* function is defined on top of *onCxLayout* (analogous to *onLayout* from Section 2.2), adding and removing the *CxLayout* constructor.

3.1 Examples

The examples in Section 2.3 all work as before. In addition, Figure 2 shows three variations, as defined below.

```
shoppingFlip  = flipLayout shopping
shoppingLR    = leftToRight shopping
shoppingTLR = leftToRight fruit <**> total
```

4. UIs with unique notfication

Note: Wolfgang Jeltsch pointed out that this optimization described in this section is not necessary.[9] The reason is that sources are never accessible to clients of the *UI* or *UI′* types, and the abstraction never replicates the sources it creates. Thus the problem I'm trying to avoid cannot happen anyway. So, I don't recommend reading this section.

As an optimization, we next switch to notifier representation in Section 1.6 for non-redundant notification. Relative to Section 3, the new *UI* representation adds a means of generating unique tags and uses *USource* in place of *Source*.

```
type UI′ a = NewTag → Win → IO (CxLayout, USource a)
type UI =
  (→) NewTag ∘ (→) Win ∘ IO ∘ (,) CxLayout ∘ USource
```

Running a UI works as in Section 3, except that *newNewTag* (Section 1.6) is invoked to make a tag generator to pass in.

```
runNamedUI :: String → UI (IO ()) → IO ()
runNamedUI name ui = start $
  do f             ← frame [visible := False, text := name]
     newTag        ← newNewTag
     win           ← panel f []
     (cxl, msrc) ← unUI ui newTag win
     set win [layout := unCxMonoid cxl (empty, above)]
     set f   [layout := hwidget win, visible := True]
     runDD (fromUSource msrc)
```

The only changes in widget creation use (a) use of the passed in tag generator to make a unique tag and (b) conversion to an *USource*.

```
iwidget mkWid attr initial = ui $ λnewTag win →
  do wid ← mkWid win [attr := initial]
     tag ← newTag
     src ← fmap (toUSource tag) (attrSource wid attr)
     return (widgetCXL wid, src)
```

References

[1] D. Leijen. wxHaskell – a portable and concise GUI library for Haskell. In *ACM SIGPLAN Haskell Workshop (HW'04)*. ACM Press, Sept. 2004.

[2] C. McBride and R. Paterson. Applicative programming with effects. To appear in Journal of Functional Programming.

---

[9] `http://haskell.org/haskellwiki/Talk:Applicative_data-driven_programming`