Denotational design with type class morphisms

Conal Elliott
LambdaPix
conal@conal.net

Abstract

Type classes provide a mechanism for varied implementations of standard interfaces. Many of these interfaces are founded in mathematical tradition and so have regularity not only of *types* but also of *properties* (laws) that must hold. Types and properties give strong guidance to the library implementor, while leaving freedom as well. Some of this remaining freedom is in *how* the implementation works, and some is in *what* it accomplishes.

To give additional guidance to the *what*, without impinging on the *how*, this paper proposes a principle of *type class morphisms* (TCMs), which further refines the compositional style of denotational semantics. The TCM idea is simply that *the instance's meaning follows the meaning's instance*. This principle determines the meaning of each type class instance, and hence defines correctness of implementation. It also serves to transfer laws about a type's semantic model, such as the class laws, to hold for the type itself. In some cases, it provides a systematic guide to implementation, and in some cases, valuable design feedback.

The paper is illustrated with several examples of types, meanings, and morphisms.

1. Introduction

Data types play a central role in structuring our programs, whether checked statically or dynamically. Data type *representations* collect related pieces of information making its use more convenient. Adding data *abstraction* gives a clean separation between a type's interface and its implementation. The ideal abstraction is as simple as possible, revealing everything the users need, while shielding them from implementation complexity. This shield benefits the implementor as well, who is free to improve hidden implementation details later, without breaking existing uses.

What kind of a thing is an *interface* that can connect implementors with users while still serving the distinct needs of each?

Part of the answer, which might be named the "form", is a collection of names of data types and names and types of operations that work on them. For instance, for a finite map, the interface may include operations for creation, insertion, and query:

$$\begin{array}{l}\textbf{abstract type}\ \mathit{Map} :: * \to * \to * \\ \mathit{empty} :: (\ldots) \qquad \Rightarrow \mathit{Map}\ k\ v \\ \mathit{insert} :: (\ldots) \qquad \Rightarrow k \to v \to \mathit{Map}\ k\ v \to \mathit{Map}\ k\ v \\ \mathit{lookup} :: (\mathit{Eq}\ k \ldots) \Rightarrow \mathit{Map}\ k\ v \to k \to \mathit{Maybe}\ v\end{array}$$

The type signatures will eventually include additional limitations (constraints) on the type parameters. The representation of Map and the implementation of its operations are not revealed.

By itself, the *form* of the interface does not deliver the promised shield. Although it successfully *hides* implementation details, it fails to *reveal* an adequate replacement. While the implementation reveals too much information to users, the signatures provide too little. It is form without *essence*. Users of this abstraction care about what *functionality* these names have. They care what the names *mean*.

Which leads to the question: what do we mean by "mean"? What is the *essence* of a type, enlivening its form? One useful answer is given by denotational semantics. The meaning of a program data type is a type of mathematical object (e.g., numbers, sets, functions, tuples). The meaning of each operation is defined as a function from the meanings of its arguments to the meaning of its result. For instance, the meaning the type $\mathit{Map}\ k\ v$ could be partial functions from k to v. In this model, empty is the completely undefined function, insert extends a partial function, and lookup is just function application, yielding $\bot$ where undefined. The meaning of the type and of its operations can be spelled out precisely and simply, as we will see throughout this paper.

Haskell provides a way to organize interfaces via type classes (Wadler and Blott 1989; Jones 1993). Library designers can present parts of a type's interface as *instances* of standard type classes, which then provides the library's users with standard vocabularies, thus making the library more easily learned. In addition to reducing learning effort, type class instances make libraries more *useful*, thanks to other libraries that contain functionality that works across *all* instances of given type classes.

Type classes provide more than names and types of operations. They also specify *laws* to be satisfied by any instance. For example, a Monoid instance must not only define a $\emptyset$ value and a binary $(\oplus)$ operation of suitable types (called "`mempty`" and "`mappend`" in Haskell). Also, $\emptyset$ must be a left and right identity for $(\oplus)$, and $(\oplus)$ must be associative. Thus a type class determines part of the essence, as well as the form, of each of its instances.

What about the rest of the meaning of a type class instance–the semantic freedom remaining after the types and laws have had their say? This paper suggests a principle for answering that question:

> *The instance's meaning follows the meaning's instance.*

That is, the meaning of each method application is given by application of *the same method* to the meanings of the arguments. In more technical terms, the semantic function is to be a *type class morphism*, i.e., it preserves class structure.

For example, suppose we want to define a Monoid instance for our finite map type. The type class morphism (TCM) principle tells us that the meaning of $\emptyset$ on maps must be $\emptyset$ for partial functions, and the meaning of $(\oplus)$ of two maps is the $(\oplus)$ of the partial functions denoted by those maps.

This paper provides several examples of types and their meanings. Sometimes the TCM property fails, and when it does, examination of the failure leads to a simpler and more compelling design for which the principle holds. In each case, class laws are guaranteed to hold, thanks to the morphisms, and so need not be proved specifically. The laws thus come, if not "for free", then "already paid for". Often, for a simple and fitting semantic model, someone already has done the hard work of verifying class laws, so the library designer can smile and accept the gift. When the model is unusual or complex enough that the work has not been done, it may be that either the model can be improved, or someone can do the hard work for the model and share it with others. If it is true, as I believe, that good semantic models tend to be reusable, then there is leverage to be gained from TCMs.

Adopting the discipline illustrated in this paper requires additional up-front effort in clarity of thinking, just as static typing does. The reward is that the resulting designs are simple and general, and sometimes have the feel of profound *inevitability*—as something beautiful we have discovered, rather than something functional we have crafted. A gift from the gods.

2. Denotational semantics and data types

Denotational semantics is a *compositional* style for precisely specifying the meanings of languages, invented by Christopher Strachey and Dana Scott in the 1960s (Scott and Strachey 1971; Stoy 1977). The idea is to specify, for each syntactic category $\mathcal{C}$,

- a mathematical *model* $[\![\mathcal{C}]\!]$ of *meanings*, and
- a semantic function $[\![\cdot]\!]_{\mathcal{C}} :: \mathcal{C} \to [\![\mathcal{C}]\!]$.

Moreover, the various $[\![\cdot]\!]_{\mathcal{C}}$ must be *compositional*, i.e.,must be defined by (mutual) structural recursion. Notationally, "$[\![\cdot]\!]_{\mathcal{C}}\ c$" is shortened to "$[\![c]\!]_{\mathcal{C}}$", and when unambiguous, "$[\![\cdot]\!]_{C}$" will be shortened to "$[\![\cdot]\!]$".

Similarly, we can apply denotational semantics to *data types within a language* to give precise meaning, independent of its implementation. Doing so yields insight into the essence of a type, which may lead to its improvement, or to informed choices about its use. It also supports clear reasoning while implementing or using the type.

As an example, consider a data type of finite maps from keys to values. The type is abstract, available only through an explicit interface that does not reveal its representation. A central question is "What does a map *mean*?" In other words, "A map is a representation of *what* mathematical object?" A simple first answer is a partial function from values to keys. Next, we'll want to be clear about the sense of "partial" here. When a map is queried for a missing key, we might want the map to yield an error (semantically $\bot$). More likely, we'll want an indication that allows for graceful recovery. For this reason, it's appealing to model a map's partiality via a success/failure type, for which Haskell's $Maybe$ type is well suited. Thus a workable model for maps is

$$[\![Map\ k\ v]\!] = k \to Maybe\ v$$

For a language, the meaning function is defined recursively over the abstract syntactic constructors. This practice transfers directly to algebraic data types. We are doing something different, however, which is to define the meaning of an *interface*, without any reference to representation.

Return now to the interface from Section 1:

$$\begin{aligned}
&\textbf{abstract type}\ Map :: * \to * \to *\\
&empty :: (...) \qquad \Rightarrow Map\ k\ v\\
&insert :: (...) \qquad \Rightarrow k \to v \to Map\ k\ v \to Map\ k\ v\\
&lookup :: (Eq\ k ...) \Rightarrow Map\ k\ v \to k \to Maybe\ v
\end{aligned}$$

The omitted additional type constraints contribute only to the implementation and not the semantics. (With $Ord\ k$ for $insert$ and $lookup$, the implementation can use a balanced tree representation (Adams 1993).)

A semantics for Map then consists of a model, as given above, and an interpretation for each member of the interface.

$$\begin{aligned}
&[\![\cdot]\!] :: Map\ k\ v \to (k \to Maybe\ v)\\
&[\![empty]\!] \qquad\quad = \lambda k \to Nothing\\
&[\![insert\ k'\ v\ m]\!] = \lambda k \to \textbf{if}\ k \equiv k'\ \textbf{then}\ Just\ v\ \textbf{else}\ [\![m]\!]\ k\\
&[\![lookup\ m\ k]\!] \quad = [\![m]\!]\ k
\end{aligned}$$

Many more functions on maps can be defined semantically within this model, such as a left-biased "union":

$$\begin{aligned}
&union_L :: (...) \Rightarrow Map\ k\ v \to Map\ k\ v \to Map\ k\ v\\
&[\![ma\ `union_L`\ mb]\!] = \lambda k \to [\![ma]\!]\ k\ `mbLeft`\ [\![mb]\!]\ k\\
&mbLeft :: Maybe\ v \to Maybe\ v \to Maybe\ v\\
&mbLeft\ (Just\ v)\ _ \quad = Just\ v\\
&mbLeft\ Nothing\ mb' = mb'
\end{aligned}$$

Other functions can be defined in terms of simpler functions. For instance, a map defined for only a single key is semantically equivalent to one made via $empty$ and $insert$.

$$\begin{aligned}
&singleton :: (...) \Rightarrow k \to v \to Map\ k\ v\\
&singleton\ k\ v \equiv insert\ k\ v\ empty
\end{aligned}$$

Given this relationship, an explicit semantics for $singleton$ would be redundant.

What is the meaning of equality (i.e., $(\equiv)$) here? The answer used throughout this paper is that *equality is semantic*, i.e.,

$$a \equiv b \iff [\![a]\!] \equiv [\![b]\!]$$

We can use $lookup$ to find out whether a key has a corresponding value, i.e., whether the key is in the map's domain. Suppose we want to know what the domain is without having to query? Maybe we want to know the size of the map's domain (number of defined keys).

$$\begin{aligned}
&size :: Map\ k\ v \to Integer\\
&[\![size\ m]\!] = |\{k \mid [\![m]\!]\ k \not\equiv Nothing\}|
\end{aligned}$$

This semantics does not give much hint of how to implement $size$.

3. Simplicity

Most of the time, we programmers reason casually and informally about our programs, and such reasoning is useful. Sometimes we want to make sure that our casual reasoning is valid, so precision tools are useful. For instance, an important aspect of software design is *simplicity* of use. Specifying semantics precisely gives us a precise way to measure and compare simplicity of designs.[1] Moreover, and of great practical importance, simpler designs are easier to reason about, giving us the opportunity to reason further and more deeply. Benefits to software include wider applicability, easier use, and better implementation performance.

[1] In my experience, simplicity often brings generality along with it. When I design an interface, I usually have in mind some things I want to do with it and how I want it to behave for those things. A complex design is able to address each of those uses and desired behaviors specifically. A simple design, however, requires achieving these specific goals without correspondingly specific design features. Instead, the design must make do with less specifics, by exploiting more general principles. As a result, unforeseen uses are more likely to be covered as well.

The model we used for maps is fairly simple, $k \to \mathit{Maybe}\ v$. Can we simplify it further? It only has two parts, $(\to)$ and Maybe. The function part is essential to what a map is, while the Maybe part is an encoding of partiality.

One way to simplify our model is to keep just the function part, and factor out the Maybe. Think of $k \to \mathit{Maybe}\ v$ not as a *partial* map from k to v, but as a *total* map from k to $\mathit{Maybe}\ v$. Then simplify and *generalize* to total maps to arbitrary types.[2]

$$\textbf{abstract type}\ \mathit{TMap}\ k\ v \quad \text{-- total map}$$
$$[\![\mathit{TMap}\ k\ v]\!] = k \to v$$

The interface for Map doesn't quite fit TMap, so let's make some changes:

$$\begin{array}{ll} \mathit{constant} :: (\ldots) & \Rightarrow v \to \mathit{TMap}\ k\ v \\ \mathit{update} :: (\ldots) & \Rightarrow k \to v \to \mathit{TMap}\ k\ v \to \mathit{TMap}\ k\ v \\ \mathit{sample} :: (\mathit{Eq}\ k\ldots) & \Rightarrow \mathit{TMap}\ k\ v \to k \to v \end{array}$$

- In place of an empty Map, we can have a constant TMap.
- Instead of inserting a key/value pair (possibly overwriting), we'll update a key (definitely overwriting).
- Sampling a map to a key always has a defined value.

As one might guess, simplifying the semantic model also simplifies the semantic function:

$$\begin{array}{ll} [\![\cdot]\!] :: \mathit{TMap}\ k\ v \to (k \to v) & \\ [\![\mathit{constant}\ v]\!] & = \lambda k \to v \\ [\![\mathit{update}\ k'\ v\ m]\!] & = \lambda k \to \textbf{if}\ k \equiv k'\ \textbf{then}\ v\ \textbf{else}\ [\![m]\!]\ k \\ [\![\mathit{sample}\ m\ k]\!] & = [\![m]\!]\ k \end{array}$$

We can mimic partial maps in terms of total maps:

$$\begin{array}{ll} \textbf{type}\ \mathit{Map}\ k\ v = \mathit{TMap}\ k\ (\mathit{Maybe}\ v) & \\ \mathit{empty} & = \mathit{constant}\ \mathit{Nothing} \\ \mathit{insert}\ k'\ v\ m & = \mathit{update}\ k'\ (\mathit{Just}\ v)\ m \\ \mathit{lookup}\ m\ k & = \mathit{sample}\ m\ k \end{array}$$

What can a union_L of total maps mean? For Map, union_L had to handle the *possibility* that both maps assign a value to the same key. With TMap, there will *always* be conflicting bindings. The conflict resolution strategy for Map is specific to Maybe. We can simplify the design by passing a combining function:

$$\begin{array}{l} \mathit{unionWith}\ ::\ (\ldots) \Rightarrow (a \to b \to c) \\ \qquad\qquad \to \mathit{TMap}\ k\ a \to \mathit{TMap}\ k\ b \to \mathit{TMap}\ k\ c \\ [\![\mathit{unionWith}\ f\ ma\ mb]\!] = \lambda k \to f\ ([\![ma]\!]\ k)\ ([\![mb]\!]\ k) \end{array}$$

This $\mathit{unionWith}$ function is simpler (semantically) than union_L on Map. As the type shows, it is also more *general*, allowing maps of different value types.

Some of our gained simplicity is illusory, since now the client of $\mathit{unionWith}$ must provide a definition of f. However, we can provide functionality that is as easy to use, using our definition of $\mathit{Map}\ k\ v$ as $\mathit{TMap}\ k\ (\mathit{Maybe}\ v)$.

$$\begin{array}{l} \mathit{union}_L :: (\ldots)\ \mathit{Map}\ k\ v \to \mathit{Map}\ k\ v \to \mathit{Map}\ k\ v \\ \mathit{union}_L \equiv \mathit{unionWith}\ \mathit{mbLeft} \end{array}$$

This definition of union_L is semantically equivalent to the one in Section 2.

4. Type classes

Type classes provide a handy way to package up parts of an interface via a standard vocabulary. Typically, a type class also has an an associated collection of rules that must be satisfied by instances of the class.

For instance, Map can be made an instance of the Monoid class, which is

$$\begin{array}{ll} \textbf{class}\ \mathit{Monoid}\ o\ \textbf{where} & \\ \quad \emptyset & :: o \\ \quad (\oplus) & :: o \to o \to o \end{array}$$

Haskell's `mempty` and `mappend` are $\emptyset$ and $(\oplus)$, respectively. The required laws are identity and associativity:

$$\begin{array}{ll} a \oplus \emptyset & \equiv a \\ \emptyset \oplus b & \equiv b \\ a \oplus (b \oplus c) & \equiv (a \oplus b) \oplus c \end{array}$$

Is our type $\mathit{Map}\ \ a\ \ b$ of partial maps a monoid? To answer "yes", we'll have to find definitions for $\emptyset$ and $(\oplus)$ that satisfy the required types and properties. The types suggest a likely guess:

$$\begin{array}{ll} \textbf{instance}\ \mathit{Monoid}\ (\mathit{Map}\ k\ v)\ \textbf{where} & \\ \quad \emptyset & = \mathit{empty} \\ \quad (\oplus) & = \mathit{union}_L \end{array}$$

It's straightforward to show that Map satisfies the monoid laws (Elliott 2009b). The crucial lemmas are that $\mathit{Nothing}$ is a left and right identity for mbLeft and mbLeft is associative.

What about TMap? Since total maps assign a value for every key, $\emptyset$ will have to come up with a value from no information. Similarly, $(\oplus)$ will have to combine the two values it finds, without being given a specific combination function. Where do we get the empty value and the combining function? From another monoid.

$$\begin{array}{ll} \textbf{instance}\ \mathit{Monoid}\ v \Rightarrow \mathit{Monoid}\ (\mathit{TMap}\ k\ v)\ \textbf{where} & \\ \quad \emptyset & = \mathit{constant}\ \ \emptyset \\ \quad (\oplus) & = \mathit{unionWith}\ (\oplus) \end{array}$$

This instance satisfies the monoid laws, which is easier to show than for Map. The simplified reasoning testifies to the value of the simpler semantic model.

However, there is a subtle problem with these two Monoid instances. Can you spot it? (Warning: Spoiler ahead.)

Section 3 defined Map in terms of TMap. Given that definition, the Monoid instance for TMap imposes a Monoid for the more specific Map type. Are these two instances for Map consistent?

Look at the standard Monoid instance for Maybe values:

$$\begin{array}{ll} \textbf{instance}\ \mathit{Monoid}\ o \Rightarrow \mathit{Monoid}\ (\mathit{Maybe}\ o)\ \textbf{where} & \\ \quad \emptyset & = \mathit{Nothing} \\ \quad \mathit{Just}\ a\ \ \oplus \mathit{Nothing} & = \mathit{Just}\ a \\ \quad \mathit{Nothing} \oplus \mathit{Just}\ b & = \mathit{Just}\ b \\ \quad \mathit{Just}\ a\ \ \oplus \mathit{Just}\ b & = \mathit{Just}\ (a \oplus b) \end{array}$$

Since $\mathit{constant}\ \mathit{Nothing} \equiv \mathit{constant}\ \emptyset$, the two definitions of $\emptyset$ for Map agree. So far, so good. Next, recall the combining function used to define union_L on partial maps:

$$\begin{array}{ll} \mathit{mbLeft} :: \mathit{Maybe}\ v \to \mathit{Maybe}\ v \to \mathit{Maybe}\ v & \\ \mathit{mbLeft}\ (\mathit{Just}\ v)\ \_ & = \mathit{Just}\ v \\ \mathit{mbLeft}\ \mathit{Nothing}\ mb' & = mb' \end{array}$$

This combining function gives union_L its left bias. When we define $\mathit{Map}\ k\ v = \mathit{TMap}\ k\ (\mathit{Maybe}\ v)$, two changes result: the left bias is lost, and a constraint is added that v must be a monoid. So, no, the imposed Monoid instance for Map is *not* consistent with its desired, original meaning.

We could eliminate this inconsistency by defining Map from scratch rather than via TMap. This solution comes at the cost of added complexity, overall. Fortunately, there is another solution that improves rather than degrades simplicity: *fix* the semantic model for Map.

[2] For simplicity of presentation, this paper does not use the more strictly compositional form: $[\![\mathit{TMap}\ k\ v]\!] = [\![k]\!] \to [\![v]\!]$.

The source of the inconsistency is that *Maybe* has a *Monoid* instance, but it's the wrong one. It's unbiased where we want left-biased. The type's *representation* suits our needs, but its associated instance does not. So, let's keep the representation while changing the type.

The *Monoid* library (Gill et al. 2001) provides some types for this sort of situation, including two wrappers around *Maybe*:

```
newtype First a = First (Maybe a)
  deriving (Eq, Ord, Read, Show, Functor, ...)
newtype Last a = Last (Maybe a)
  deriving (Eq, Ord, Read, Show, Functor, ...)
```

The sole purpose of these type wrappers is to replace *Maybe*'s unbiased *Monoid* instance with a left-biased (*First*) or right-biased (*Last*) instance.

```
instance Monoid (First a) where
  ∅ = First Nothing
  l@(First (Just _)) ⊕ _ = l
  First Nothing ⊕ r      = r
instance Monoid (Last a) where
  ∅ = Last Nothing
  _ ⊕ r@(Last (Just _)) = r
  l ⊕ Last Nothing      = l
```

This $(\oplus)$ for *First* is just like the function *mbLeft* used for defining $union_L$ and hence $(\oplus)$ for *Map*. We can solve our consistency problem by replacing *Maybe* with *First* in *Map*:

```
type Map k v = TMap k (First v)
```

With this change, the *Monoid* instance for *Map* can disappear completely. The desired behavior falls out of the *Monoid* instances for *TMap* and *First*.

This definition also translates to a new semantic model for partial maps:

$$[\![Map\ k\ v]\!] = k \to First\ v$$

This new version hints of the left-bias of $(\oplus)$. It also gives a single point of change if we decide we prefer a right-bias: replace *First* with *Last*.

5. Type class morphisms

Section 2 illustrated how denotational semantics gives precise meaning to data types, which serves as a basis for casual and precise reasoning. Simplicity of semantic models (Section 3) improves generality and ease of reasoning. Section 4 then brought in type classes, which give a consistent way to structure interfaces. The semantic models of instances give a basis for proving the associated class properties.

Now we're ready to get to the heart of this paper, which is a synergy of the two disciplines of denotational semantics and type classes.

5.1 Monoid

Consider again the *Monoid* instance for total maps:

```
instance Monoid v ⇒ Monoid (TMap k v) where
  ∅     = constant    ∅
  (⊕) = unionWith (⊕)
```

Now recast it into semantic terms, with a "semantic instance", specifying the meaning, rather than the implementation, of a type class instance:

$$\begin{array}{l}\textbf{instance}_{\textbf{sem}}\ Monoid\ v \Rightarrow Monoid\ (TMap\ k\ v)\ \textbf{where}\\ \quad [\![\emptyset]\!] = [\![constant\quad \emptyset]\!]\\ \quad [\![(\oplus)]\!] = [\![unionWith\ (\oplus)]\!]\end{array}$$

The meanings of *constant* and *unionWith* are given in Section 3:

$$\begin{array}{ll}[\![constant\ v]\!] & \equiv \lambda k \to v\\ [\![unionWith\ f\ ma\ mb]\!] & \equiv \lambda k \to f\ ([\![ma]\!]\ k)\ ([\![mb]\!]\ k)\end{array}$$

Substituting, we get

$$\begin{array}{l}\textbf{instance}_{\textbf{sem}}\ Monoid\ v \Rightarrow Monoid\ (TMap\ k\ v)\ \textbf{where}\\ \quad [\![\emptyset]\!] = \lambda k \to \emptyset\\ \quad [\![ma \oplus mb]\!] = \lambda k \to [\![ma]\!]\ k \oplus [\![mb]\!]\ k\end{array}$$

The RHSs (right-hand sides) of these semantic definitions resemble an instance in the *Monoid* library:

```
instance Monoid b ⇒ Monoid (a → b) where
  ∅     = λa → ∅
  f ⊕ g = λa → f a ⊕ g a
```

This instance lets us simplify the *TMap* semantic monoid instance:

$$\begin{array}{l}\textbf{instance}_{\textbf{sem}}\ Monoid\ v \Rightarrow Monoid\ (TMap\ k\ v)\ \textbf{where}\\ \quad [\![\emptyset]\!] = \emptyset\\ \quad [\![ma \oplus mb]\!] = [\![ma]\!] \oplus [\![mb]\!]\end{array}$$

This property of $[\![\cdot]\!]$ has a special name in mathematics: $[\![\cdot]\!]$ is a *monoid homomorphism*, or more tersely a "monoid morphism".

Note what has happened here. The semantics of the *Monoid* part of the *TMap* interface is completely specified by saying that $[\![\cdot]\!]$ is a morphism for that class.

Besides succinctness, why else might we care that a meaning function is a monoid morphism? Because this property means that, at least for the *Monoid* part of the interface, *the type behaves like its model*. As users, we can therefore think of the type as *being* its model. Functions are well-studied, so the experience we have with the model is a great help in thinking about the type. Our reasoning is shielded from the implementation complexity. The flip side is that we are cut off from reasoning about its operational performance.

What about other type classes, besides *Monoid*?

5.2 Functor

The *Functor* interface is

```
class Functor f where
  fmap :: (a → b) → f a → f b
```

with laws:

$$\begin{array}{ll}fmap\ id & \equiv id\\ fmap\ (h \circ g) & \equiv fmap\ h \circ fmap\ g\end{array}$$

A total map can be made an instance of *Functor*. Mapping a function over a total map applies the function to each value in a key/value binding:

$$\begin{array}{l}\textbf{instance}_{\textbf{sem}}\ Functor\ (TMap\ k)\ \textbf{where}\\ \quad [\![fmap\ f\ m]\!] = \lambda k \to f\ ([\![m]\!]\ k)\end{array}$$

This semantic definition for *fmap* satisfies the *compositional* discipline of denotational semantics, in that the meaning of $fmap\ f\ m$ is defined in terms of the meaning of its *TMap* argument, m. Moreover, this semantics also satisfies the *Functor* laws (Elliott 2009b).

Now let's play with this definition. Write the RHS in point-free form, using function composition:

$$[\![fmap\ f\ m]\!] = f \circ [\![m]\!]$$

Next, note the *Functor* instance for functions:

```
instance Functor ((→) a) where fmap = (∘)
```

Our *fmap* semantics then become

$$[\![fmap\ f\ m]\!] = fmap\ f\ [\![m]\!]$$

This property of $[\![\cdot]\!]$, often written as $[\![\cdot]\!] \circ fmap\ f \equiv fmap\ f \circ [\![\cdot]\!]$, also has special name in mathematics: $[\![\cdot]\!]$ is a *natural transformation*, also called a "morphism on functors" (Mac Lane 1998) or (for consistency) a "functor morphism".

5.3 What is a type class morphism?

The monoid and functor morphism properties say that the method structure is preserved. That is, the interpretation of a method application on a type T is *the same method* applied to the interpretations of the type-T arguments.

As a design guide, I recommend that any semantic function for a type T (a) be as simple as possible without loss of precision, and (b) be a type class morphism (TCM) with respect to each of the type classes it implements. Consequently, users can think of the type as *being* its (simple) model, rather than being an implementation of the model.

Another way to describe the TCM property is:

The instance's meaning follows the meaning's instance.

In other words, *meaning preserves class structure*, by preserving the structure of each method application, in ways suited to the type of each method. See Sections 5.1 and 5.2 for examples.

5.4 Applicative functor

Let's try one more. The $Applicative$ (applicative functor) type class has more structure than $Functor$, but less than $Monad$. The interface:

$$\begin{array}{l}\textbf{infixl}\ 4\ \circledast \\ \textbf{class}\ Functor\ f \Rightarrow Applicative\ f\ \textbf{where} \\ \quad pure :: a \to f\ a \\ \quad (\circledast)\ :: f\ (a \to b) \to f\ a \to f\ b\end{array}$$

As usual, this class also comes with collection of laws (McBride and Paterson 2008).

This time, instead of thinking about what $Applicative$ instance we might want for $TMap$, let's see if the TCM property will tell us. Suppose that $TMap$ is an *applicative morphism*, i.e.,

$$\begin{array}{l}\textbf{instance}_{\textbf{sem}}\ Applicative\ (TMap\ k)\ \textbf{where} \\ \quad [\![pure\ b]\!] = pure\ b \\ \quad [\![mf \circledast mx]\!] = [\![mf]\!] \circledast [\![mx]\!]\end{array}$$

Indeed, there is an $Applicative$ instance for functions, where $pure$ and $(\circledast)$ are the classic K and S combinators:

$$\begin{array}{l}\textbf{instance}\ Applicative\ ((\to)\ a)\ \textbf{where} \\ \quad pure\ b = \lambda a \to b \\ \quad h \circledast g\ = \lambda a \to (h\ a)\ (g\ a)\end{array}$$

Substituting the semantic instance ($Applicative$ on functions) in the RHS of our $Applicative$ instance for $TMap$ above yields

$$\begin{array}{l}\textbf{instance}_{\textbf{sem}}\ Applicative\ (TMap\ k)\ \textbf{where} \\ \quad [\![pure\ v]\!] = \lambda k \to v \\ \quad [\![mf \circledast mx]\!] = \lambda k \to ([\![mf]\!]\ k)\ ([\![mx]\!]\ k)\end{array}$$

In other words, a pure map is one that maps all keys to the same value; and "application" of a map full of functions to a map full of arguments is defined pointwise.

Often $(\circledast)$ is used via n-ary lifting functions:

$$\begin{array}{l}liftA_2\ h\ u\ v\ = pure\ h \circledast u \circledast v \\ liftA_3\ h\ u\ v\ w = pure\ h \circledast u \circledast v \circledast w \\ \qquad\qquad ...\end{array}$$

For functions, simplifying the RHS of $liftA_2$ gives

$$liftA_2\ h\ u\ v \equiv \lambda x \to h\ (u\ x)\ (v\ x)$$

and similarly for $liftA_3$.

With these definitions in mind, let's look again at two of our functions on total maps, from Section 3

$$\begin{array}{l}constant\ \ :: (...) \Rightarrow v \to TMap\ k\ v \\ unionWith\ :: (...) \Rightarrow (a \to b \to c) \\ \qquad\qquad \to TMap\ k\ a \to TMap\ k\ b \to TMap\ k\ c \\ [\![constant\ v]\!] \equiv \lambda k \to v \\ [\![unionWith\ f\ ma\ mb]\!] \equiv \lambda k \to f\ ([\![ma]\!]\ k)\ ([\![mb]\!]\ k)\end{array}$$

Now we see that $constant$ and $unionWith$ are just synonyms for $pure$ and $liftA_2$, so we can eliminate the former terms (and their definitions) in favor of the $Applicative$ ones. This change gives us more for less: our type's specification is simpler; and it can be used with libraries that work with applicative functors.

We have a last question to answer: Does our $Applicative$ instance satisfy the required laws?

6. Class laws

Section 5 showed that $TMap$ satisfies the $Monoid$ and $Functor$ laws. Does $TMap$ also satisfy the $Applicative$ laws? We'll now see that the answer is "yes", simply because $TMap$ is (by construction) an applicative morphism.

Consider, for instance, the *composition law*.

$$pure\ (\circ) \circledast u \circledast v \circledast w \equiv u \circledast (v \circledast w)$$

We are defining equality *semantically*, i.e., $u \equiv v \iff [\![u]\!] \equiv [\![v]\!]$, so the composition law is equivalent to

$$[\![pure\ (\circ) \circledast u \circledast v \circledast w]\!] \equiv [\![u \circledast (v \circledast w)]\!]$$

Working left to right,

$$\begin{array}{ll} & [\![pure\ (\circ) \circledast u \circledast v \circledast w]\!] \\ \equiv & \{\ [\![\cdot]\!] \text{ is an applicative morphism }\} \\ & [\![pure\ (\circ)]\!] \circledast [\![u]\!] \circledast [\![v]\!] \circledast [\![w]\!] \\ \equiv & \{\ [\![\cdot]\!] \text{ is an applicative morphism }\} \\ & pure\ (\circ) \circledast [\![u]\!] \circledast [\![v]\!] \circledast [\![w]\!] \\ \equiv & \{\ \text{composition law on semantics }\} \\ & [\![u]\!] \circledast ([\![v]\!] \circledast [\![w]\!]) \\ \equiv & \{\ [\![\cdot]\!] \text{ is an applicative morphism }\} \\ & [\![u]\!] \circledast [\![v \circledast w]\!] \\ \equiv & \{\ [\![\cdot]\!] \text{ is an applicative morphism }\} \\ & [\![u \circledast (v \circledast w)]\!]\end{array}$$

The proofs for the other laws follow in exactly the same way.

These proofs make no use of the specifics of the $Applicative$ instance for $TMap$. They are, therefore, applicable to *every* type for which equality is defined by an applicative morphism (e.g., $[\![\cdot]\!]$).

Similar reasoning and conclusions hold for the $Monoid$, $Functor$, $Monad$, and $Arrow$ classes (Elliott 2009b).

7. Type composition

Trying the TCM game with Map in place of $TMap$ reveals a problem. The $Monoid$ case comes out fine, but let's see what happens with $Functor$. First, we'll have to make Map abstract again (not a synonym) so that we can define a $Functor$ instance.

$$\begin{array}{l}\textbf{abstract type}\ Map \\ [\![Map\ k\ v]\!] = k \to First\ v\end{array}$$

Mapping a function over partial map modifies the defined values:

$$\begin{array}{l}\textbf{instance}_{\textbf{sem}}\ Functor\ (Map\ k)\ \textbf{where} \\ \quad [\![fmap\ f\ m]\!] = \lambda k \to fmap\ f\ ([\![m]\!]\ k)\end{array}$$

The RHS *fmap* uses the *Functor* for *First* (derived from *Maybe*).

Now if we simplify and re-arrange as with $TMap$, we get

$$\begin{array}{l}\textbf{instance}_{\textbf{sem}}\ \mathit{Functor}\ (\mathit{Map}\ k)\ \textbf{where}\\ \quad [\![\mathit{fmap}\ f\ m]\!] = \mathit{fmap}\ (\mathit{fmap}\ f)\ [\![m]\!]\end{array}$$

So $[\![\cdot]\!]$ is *not* a *Functor* morphism.

A similar difficulty arises with $Applicative$. The map $pure\ v$ has value v for every key. For a function-valued map mf and an argument-valued map mx, the map $mf \circledast mx$ assigns a defined value of $f\ x$ to a key k if f is the defined value of mf at k and x is the defined value of mx at k. If mf or mx is undefined at k then so is $mf \circledast mx$. This meaning is formalized in the following $Applicative$ instance, which uses the $Applicative$ instance for *First* (derived from *Maybe*):

$$\begin{array}{l}\textbf{instance}_{\textbf{sem}}\ \mathit{Applicative}\ (\mathit{Map}\ k)\ \textbf{where}\\ \quad [\![\mathit{pure}\ v]\!] = \lambda k \to \mathit{pure}\ v\\ \quad [\![mf \circledast mx]\!] = \lambda k \to [\![mf]\!]\ k \circledast [\![mx]\!]\ k\end{array}$$

Using the $Applicative$ instance for functions, this instance becomes

$$\begin{array}{l}\textbf{instance}_{\textbf{sem}}\ \mathit{Applicative}\ (\mathit{Map}\ k)\ \textbf{where}\\ \quad [\![\mathit{pure}\ v]\!] = \mathit{pure}\ (\mathit{pure}\ v)\\ \quad [\![mf \circledast mx]\!] = \mathit{liftA}_2\ (\circledast)\ [\![mf]\!]\ [\![mx]\!]\end{array}$$

Thus $[\![\cdot]\!]$ is also *not* an $Applicative$ morphism.

Again, these failures look like bad news. Must we abandon the TCM principle, or does the failure point us to a new, and possibly better, model for Map, as in Section 4?

The rewritten semantic instances above do not make use of any properties of Map other than being a composition of two functors for the first instance and two applicative functors for the second. So let's generalize. As McBride and Paterson (2008) observed, $Functor$ and $Applicative$ are closed under composition:

$$\begin{array}{l}\textbf{newtype}\ (h \circ g)\ a = O\ (h\ (g\ a))\\ \textbf{instance}\ (\mathit{Functor}\ h, \mathit{Functor}\ g)\\ \qquad \Rightarrow \mathit{Functor}\ (h \circ g)\ \textbf{where}\\ \quad \mathit{fmap}\ f\ (O\ hga) = O\ (\mathit{fmap}\ (\mathit{fmap}\ f)\ hga)\\ \textbf{instance}\ (\mathit{Applicative}\ h, \mathit{Applicative}\ g)\\ \qquad \Rightarrow \mathit{Applicative}\ (h \circ g)\ \textbf{where}\\ \quad \begin{array}{ll}\mathit{pure}\ a & = O\ (\mathit{pure}\ (\mathit{pure}\ a))\\ O\ hgf \circledast O\ hgx & = O\ (\mathit{liftA}_2\ (\circledast)\ hgf\ hgx)\end{array}\end{array}$$

Define the meaning of a composition to be a composition:

$$\begin{array}{ll}[\![(h \circ g)\ t]\!] & = ([\![h]\!] \circ g)\ t \quad \text{-- types}\\ [\![O\ hga]\!]_{h \circ g} & = O\ [\![hga]\!]_h \quad \text{-- values}\end{array}$$

If $[\![\cdot]\!]_h$ is a *Functor* morphism, then $[\![\cdot]\!]_{h \circ g}$ is also a *Functor* morphism, and similarly for $Applicative$ (Elliott 2009b).

Now redefine Map:

$$\textbf{type}\ \mathit{Map}\ k \equiv \mathit{TMap}\ k \circ \mathit{First}$$

More explicitly,

$$\begin{array}{ll}[\![\mathit{Map}\ k\ v]\!] & = [\![(\mathit{TMap}\ k \circ \mathit{First})\ v]\!]\\ & \cong k \to \mathit{First}\ v\end{array}$$

These definitions say exactly what we mean, more directly than the previous definition. A (left-biased) partial map is a composition of two simpler ideas: total maps and *First* (left-biased $Maybe$). Given this new, more succinct definition, we can scrap the $Functor$ and $Applicative$ semantic instance definitions as redundant.

It may look like we've just moved complexity around, rather than eliminating it. However, type composition is a *very* reusable notion, which is why it was already defined, along with supporting proofs that the $Functor$ and $Applicative$ laws hold.

8. Deriving type class instances

Implementations of type class instances can sometimes be derived mechanically from desired type class morphisms. Suppose we have $[\![\cdot]\!] :: T \to T'$, where T' belongs to one or more classes and $[\![\cdot]\!]$ is *invertible*. Then, combining $[\![\cdot]\!]$ and $[\![\cdot]\!]^{-1}$ gives instances that satisfy the morphism properties.

For instance, consider the $Monoid$ morphism properties:

$$\begin{array}{ll}[\![\emptyset]\!] & \equiv \emptyset\\ [\![u \oplus v]\!] & \equiv [\![u]\!] \oplus [\![v]\!]\end{array}$$

Because $[\![\cdot]\!] \circ [\![\cdot]\!]^{-1} \equiv id$, these properties are satisfied if

$$\begin{array}{ll}[\![[\![\emptyset]\!]]\!]^{-1} & \equiv [\![\emptyset]\!]^{-1}\\ [\![[\![u \oplus v]\!]]\!]^{-1} & \equiv [\![[\![u]\!] \oplus [\![v]\!]]\!]^{-1}\end{array}$$

Then, because $[\![\cdot]\!]^{-1} \circ [\![\cdot]\!] \equiv id$, these properties are equivalent to the following class definition:

$$\begin{array}{l}\textbf{instance}\ \mathit{Monoid}\ T\ \textbf{where}\\ \quad \begin{array}{ll}\emptyset & = [\![\emptyset]\!]^{-1}\\ u \oplus v & = [\![[\![u]\!] \oplus [\![v]\!]]\!]^{-1}\end{array}\end{array}$$

By construction, $[\![\cdot]\!]$ is a $Monoid$ morphism. Similarly for the other type classes. Assuming the class laws hold for T', they hold as well for T, as shown in Section 6.

If $[\![\cdot]\!]$ and $[\![\cdot]\!]^{-1}$ have implementations, then we can stop here. Otherwise, or if we want to optimize the implementation, we can do some more work, to rewrite the synthesized definitions.

9. Memo tries

As an example of synthesizing type class instances, let's consider a form of memoization, based on a variant of the total maps from Section 2, namely a complete memo trie. The central idea of the function memoizer is associating a type of trie to each domain type we want to memoize over.

Following the idea of generic tries (Hinze 2000) and its implementation with associated types (Chakravarty et al. 2005), define a class of types with trie representations:[3]

$$\begin{array}{l}\textbf{class}\ \mathit{HasTrie}\ a\ \textbf{where}\\ \quad \textbf{data}\ (\twoheadrightarrow)\ a :: * \to *\end{array}$$

The type $a \twoheadrightarrow b$ represents a trie that maps values of type a to values of type b. The trie representation depends only on a.

The $HasTrie$ class also contains converters between functions and tries:

$$\begin{array}{ll}\mathit{trie} & :: (a \to b) \to (a \twoheadrightarrow b)\\ \mathit{untrie} & :: (a \twoheadrightarrow b) \to (a \to b)\end{array}$$

The $untrie$ and $trie$ methods must be an embedding-projection pair:

$$\begin{array}{l}\mathit{trie} \circ \mathit{untrie} \equiv id\\ \mathit{untrie} \circ \mathit{trie} \sqsubseteq id\end{array}$$

The reason for $(\sqsubseteq)$ is that keys get fully evaluated while building and searching the trie structure, even if they are not evaluated by the function being memoized. See (Elliott 2009b) for a development of several $HasTrie$ instances.

Given the $HasTrie$ class, memoization is trivial to implement:

$$\begin{array}{l}\mathit{memo} :: \mathit{HasTrie}\ a \Rightarrow (a \to b) \to (a \to b)\\ \mathit{memo} = \mathit{untrie} \circ \mathit{trie}\end{array}$$

[3] This formulation is based on a design from Spencer Janssen.

The second inverse property implies that memo approximates the identity function (semantically). In practice, $memo \equiv hyperstrict$, where

$$hyperstrict :: Hyper\ a \Rightarrow (a \to b) \to (a \to b)$$
$$hyperstrict\ f\ a = hyperEval\ a\ `seq`\ f\ a$$

Here I've assumed a *hyperEval* function to evaluate its argument fully. This function could either be built-in magic, or defined as a method from a type class, $Hyper$.

Multi-argument curried functions can be memoized by repeated uses of $memo$. For instance,

$$memo_2 :: (HasTrie\ a, HasTrie\ b) \Rightarrow (a \to b \to c) \to (a \to b \to c)$$
$$memo_2\ f = memo\ (memo \circ f)$$

9.1 A semantic false start

Using $[\![\cdot]\!] = untrie$ as the semantic function looks promising. However, $untrie$ has no inverse, simply because it only produces hyper-strict functions. This lack of invertibility means that the synthesis technique in Section 8 fails to apply.

Worse yet, $untrie$ cannot be a monoid morphism for *any* $Monoid$ instance of $a \twoheadrightarrow b$. To see why, assume the simplest morphism property, the $Monoid$ identity: $untrie\ \emptyset \equiv \emptyset$. Applying $trie$ to both sides, and using $trie \circ untrie \equiv id$, it must be that $\emptyset \equiv trie\ \emptyset$. It then follows that

$$\begin{aligned} untrie\ \emptyset &\equiv untrie\ (trie\ \emptyset) \\ &\equiv hyperstrict\ \emptyset \end{aligned}$$

We're okay if $hyperstrict\ \emptyset \equiv \emptyset$, but that is not the case, because $\emptyset$ on functions is $\lambda x \to \emptyset$, which is not even strict (assuming $\emptyset \not\equiv \bot$).

The conclusion that $untrie$ cannot be a monoid morphism sounds like a failure, while in fact it gives us helpful information. It says that $untrie$ cannot be our semantic function and why it cannot. The difficulty came from the presence of non-hyper-strict functions in the semantic domain, so let's remove them. Instead of modeling $tries$ as functions, model them as hyper-strict functions, which we'll write as "$a \xrightarrow{H} b$":

Hyper-strict functions can be defined purely mathematically, for use in specifications only, or they can be defined in the implementation language:

$$\mathbf{newtype}\ a \xrightarrow{H} b = H\ \{ unH :: a \to b \}$$

With the condition that the contained $a \to b$ is hyper-strict.

With this new and improved semantic domain, we can define a semantic function:

$$[\![\cdot]\!] :: (a \twoheadrightarrow b) \to (a \xrightarrow{H} b)$$
$$[\![\cdot]\!] = H \circ untrie$$

The reason we can legitimately use H here is that $untrie$ always produces hyper-strict functions.

9.2 Type class morphisms regained

The problem with our first attempt (Section 9.1) was that the semantic function was not invertible. The new semantic function, however, is invertible.

$$[\![\cdot]\!]^{-1} :: (a \xrightarrow{H} b) \to (a \twoheadrightarrow b)$$
$$[\![\cdot]\!]^{-1} = trie \circ unH$$

Because $[\![\cdot]\!]$ is invertible, we can synthesize instances for $a \twoheadrightarrow b$ for all of the classes that the semantic model ($a \xrightarrow{H} b$) inhabits, as shown in Section 8. As desired, $[\![\cdot]\!]$ is, by construction, a morphism with respect to each of those classes. See (Elliott 2009b) for a proof that $[\![\cdot]\!]$ and $[\![\cdot]\!]^{-1}$ are inverses and for the class instances for hyper-strict functions.

10. Numeric overloadings

Overloading makes it possible to use familiar and convenient numeric notation for working with a variety of types. The principle of type class morphisms can help to guide such overloadings, suggesting when to use it and how to use it correctly.

10.1 Functions

As an example, we can apply numeric notation for manipulating *functions*. Given functions f and g, what could be the meaning of $f + g$, $f * g$, $recip\ f$, $sqrt\ f$, etc? A useful answer is that the operations are applied *point-wise*, i.e.,

$$\begin{aligned} f + g &= \lambda x \to f\ x + g\ x \\ f * g &= \lambda x \to f\ x * g\ x \\ recip\ f &= \lambda x \to recip\ (f\ x) \\ sqrt\ f &= \lambda x \to sqrt\ (f\ x) \\ &\ldots \end{aligned}$$

It's also handy to add literal numbers to the mix, e.g., $5 * sqrt\ f$, which can mean $\lambda x \to 5 * sqrt\ (f\ x)$. In Haskell, numeric literals are overloaded, using $fromInteger$ and $fromRational$, so add

$$fromInteger\ n = const\ (fromInteger\ n)$$

And similarly for rational literals.

Thanks to the $Functor$ and $Applicative$ instances for functions (given in Section 5), these interpretations are equivalent to

$$\begin{aligned} fromInteger &= pure \circ fromInteger \\ (+) &= liftA_2\ (+) \\ (*) &= liftA_2\ (*) \\ recip &= fmap\ recip \\ sqrt &= fmap\ sqrt \\ &\ldots \end{aligned}$$

With this rephrasing, these definitions can be used not just with functions, but with *any* applicative functor.

This notation is convenient and pretty, but what justifies it, semantically? Looking below the notation, in what sense can functions be interpreted as numbers?

A compelling answer would be a simple function $[\![\cdot]\!]$ mapping functions to a familiar number type, such that the $[\![\cdot]\!]$ is a "numeric morphism", i.e., a morphism over all of the standard numeric classes.

Is there such a $[\![\cdot]\!]$? Yes! And better yet, there's a great wealth of them. The functions f and g above have type $a \to b$ for a numeric type b. Define a family of interpretation functions indexed over a. For all $x :: a$,

$$[\![\cdot]\!]^x :: (a \to b) \to b$$
$$[\![h]\!]^x = h\ x$$

Then every $[\![\cdot]\!]^x$ is a numeric morphism. To see why, consider two operations. The others all work out similarly.[4]

$$\begin{aligned} [\![fromInteger\ n]\!]^x &\equiv (fromInteger\ n)\ x \\ &\equiv (const\ (fromInteger\ n))\ x \end{aligned}$$

[4] There is a technical difficulty in a Haskell implementation. The Num class (and hence its descendants) are required also to implement Eq and $Show$. The methods of those classes cannot be defined such that $[\![\cdot]\!]^x$ is an Eq or $Show$ morphism, because $(\equiv)$ yields $Bool$ and $show$ yields $String$, One could perhaps fix this problem by making these classes more morphism-friendly or by eliminating them as a superclass of Num. My thanks to Ganesh Sittampalam for pointing out this issue.

$$\equiv \mathit{fromInteger}\ n$$

$$\begin{array}{ll} [\![f+g]\!]^x & \equiv (f+g)\ x \\ & \equiv f\ x + g\ x \\ & \equiv [\![f]\!]^x + [\![g]\!]^x \end{array}$$

These proofs apply whenever numeric instances are defined above, via fmap, pure, and liftA_2, and when the semantic function is a $\mathit{Functor}$ and $\mathit{Applicative}$ morphism (possibly using the identity functor) (Elliott 2009b).

10.2 Errors

For semantic simplicity, errors can be represented as $\bot$. If we want to respond to errors in a semantically tractable way (hence without resorting to IO), we'll want an explicit encoding. One choice is $\mathit{Maybe}\ a$, where $\mathit{Nothing}$ means failure and $\mathit{Just}\ s$ means success with values $s :: a$. Another choice is $\mathit{Either}\ x\ a$, where $\mathit{Left}\ \mathit{err}$ means a failure described by $\mathit{err} :: x$, while $\mathit{Right}\ s$ means success.

Programming with either explicit encoding can be quite tedious, but a few well-chosen type class instances can hide almost all of the routine plumbing of failure and success encodings. The key again is $\mathit{Functor}$ and $\mathit{Applicative}$. The following instances suffice.

$$\begin{array}{l} \textbf{instance}\ \mathit{Functor}\ (\mathit{Either}\ x)\ \textbf{where} \\ \quad \begin{array}{ll} \mathit{fmap}\ \_\ (\mathit{Left}\ x) & = \mathit{Left}\ x \\ \mathit{fmap}\ f\ (\mathit{Right}\ y) & = \mathit{Right}\ (f\ y) \end{array} \\ \textbf{instance}\ \mathit{Applicative}\ (\mathit{Either}\ x)\ \textbf{where} \\ \quad \begin{array}{ll} \mathit{pure}\ y & = \mathit{Right}\ y \\ \mathit{Left}\ x \circledast \_ & = \mathit{Left}\ x \\ \_ \circledast \mathit{Left}\ x & = \mathit{Left}\ x \\ \mathit{Right}\ f \circledast \mathit{Right}\ x & = \mathit{Right}\ (f\ x) \end{array} \end{array}$$

A simple semantics for Either forgets the error information, replacing it with $\bot$:

$$\begin{array}{l} [\![\cdot]\!] :: \mathit{Either}\ e\ a \to a \\ [\![\mathit{Left}\ \ e]\!] = \bot \\ [\![\mathit{Right}\ a]\!] = a \end{array}$$

To give $\mathit{Either}\ e\ a$ numeric instances, follow the same template as with functions (Section 10.1), e.g., $(+) = \mathit{liftA}_2\ (+)$. The numeric morphism properties hold *only* if the numeric methods on the type a are strict. For an n-ary function h that is strict in all n arguments:

$$[\![\mathit{liftA}_n\ f\ e_1\ ...\ e_n]\!] \equiv f\ [\![e_1]\!]\ ...\ [\![e_n]\!]$$

where $\mathit{fmap} \equiv \mathit{liftA}_1$. The relevance of ($\bot$-)strictness here is that $(\circledast)$ on Either yields a Left if either argument is a Left (Elliott 2009b). Assuming $(+)$ is strict then,

$$\begin{array}{ll} & [\![e + e']\!] \\ \equiv & \quad \{\ (+) \text{ on } \mathit{Either}\ x\ y\ \} \\ & [\![\mathit{liftA}_2\ (+)\ e\ e']\!] \\ \equiv & \quad \{\ (+) \text{ is strict}\ \} \\ & [\![e]\!] + [\![e']\!] \end{array}$$

The semantic property on liftA_n above is nearly a morphism property, but the liftA_n is missing on the right. We can restore it by changing the semantic domain:

$$\begin{array}{l} \textbf{newtype}\ \mathit{Id}\ a = \mathit{Id}\ a \\ [\![\cdot]\!] :: \mathit{Either}\ e\ a \to \mathit{Id}\ a \\ [\![\mathit{Left}\ \ e]\!] = \mathit{Id}\ \bot \\ [\![\mathit{Right}\ a]\!] = \mathit{Id}\ a \end{array}$$

$[\![\cdot]\!]$ is *almost* a $\mathit{Functor}$ and $\mathit{Applicative}$ morphism, but not quite, because of the possibility of non-strict functions (Elliott 2009b).

10.3 Expressions

Another use of numeric overloading is for syntax manipulation, as in compiling domain-specific embedded languages (Elliott et al. 2003; Augustsson et al. 2008). In this context, there is a parameterized type of numeric expressions, often an algebraic data type. Using *generalized* algebraic data types (Peyton Jones et al. 2006) helps with static typing.

10.3.1 Simple expressions

A simple representation for expressions replaces each method with a constructor:

$$\begin{array}{l} \textbf{data}\ E :: * \to * \ \textbf{where} \\ \quad \begin{array}{ll} \mathit{FromInteger} & :: \mathit{Num}\ a \Rightarrow \mathit{Integer} \to E\ a \\ \mathit{Add} & :: \mathit{Num}\ a \Rightarrow E\ a \to E\ a \to E\ a \\ \mathit{Mult} & :: \mathit{Num}\ a \Rightarrow E\ a \to E\ a \to E\ a \\ & ... \\ \mathit{FromRational} & :: \mathit{Fractional}\ a \Rightarrow \mathit{Rational} \to E\ a \\ \mathit{Recip} & :: \mathit{Fractional}\ a \Rightarrow E\ a \to E\ a \\ & ... \end{array} \end{array}$$

A convenient way to build up expressions is through numeric overloading, e.g.,

$$\begin{array}{l} \textbf{instance}\ \mathit{Num}\ a \Rightarrow \mathit{Num}\ (E\ a)\ \textbf{where} \\ \quad \begin{array}{ll} \mathit{fromInteger} & = \mathit{FromInteger} \\ (+) & = \mathit{Add} \\ (*) & = \mathit{Mult} \\ & ... \end{array} \\ \textbf{instance}\ \mathit{Fractional}\ a \Rightarrow \mathit{Fractional}\ (E\ a)\ \textbf{where} \\ \quad \begin{array}{ll} \mathit{fromRational} & = \mathit{FromRational} \\ \mathit{recip} & = \mathit{Recip} \\ & ... \end{array} \end{array}$$

Expressions are not just representations; they have meaning. To make that meaning precise, define a semantic function:

$$\begin{array}{ll} [\![\cdot]\!] :: E\ a \to a & \\ [\![\mathit{FromInteger}\ n]\!] & = \mathit{fromInteger}\ n \\ [\![\mathit{Add}\ \ a\ b]\!] & = [\![a]\!] + [\![b]\!] \\ [\![\mathit{Mult}\ a\ b]\!] & = [\![a]\!] * [\![b]\!] \\ & ... \\ [\![\mathit{FromRational}\ n]\!] & = \mathit{fromRational}\ n \\ [\![\mathit{Recip}\ a]\!] & = \mathit{recip}\ [\![a]\!] \\ & ... \end{array}$$

To ensure that expressions behave like what they mean, check that $[\![\cdot]\!]$ is a morphism over Num and $\mathit{Fractional}$. The definitions of $[\![\cdot]\!]$ and the instances make the proofs trivially easy. For instance,

$$[\![a + b]\!] \equiv [\![\mathit{Add}\ a\ b]\!] \equiv [\![a]\!] + [\![b]\!]$$

using the definitions of $(+)$ on E and $[\![\cdot]\!]$ on Add.

How robust are these numeric morphisms? If the representation deviates from their one-to-one correspondence with methods, will the numeric morphism properties still hold?

Let's try some variations, including self-optimizing expressions and expressions with variables.

10.3.2 Generalizing literals

As a first step, replace $\mathit{FromInteger}$ and $\mathit{FromRational}$ with a general constructor for literals.

$$\begin{array}{l} \textbf{data}\ E :: * \to * \ \textbf{where} \\ \quad \mathit{Literal} :: a \to E\ a \\ \quad ... \end{array}$$

Change the relevant methods:

$$\begin{aligned}
&\textbf{instance } \mathit{Num}\ a \Rightarrow \mathit{Num}\ (E\ a) \textbf{ where}\\
&\quad \mathit{fromInteger} = \mathit{Literal} \circ \mathit{fromInteger}\\
&\qquad\qquad \ldots\\
&\textbf{instance } \mathit{Fractional}\ a \Rightarrow \mathit{Fractional}\ (E\ a) \textbf{ where}\\
&\quad \mathit{fromRational} = \mathit{Literal} \circ \mathit{fromRational}\\
&\qquad\qquad \ldots
\end{aligned}$$

Finally, replace two $[\![\cdot]\!]$ cases with one:

$$[\![\mathit{Literal}\ x]\!] = x$$

Everything type-checks; but does it *morphism-check*? Verify the two affected methods:

$$\begin{aligned}
&[\![\mathit{fromInteger}\ n]\!]\\
\equiv\ &[\![\mathit{Literal}\ (\mathit{fromInteger}\ n)]\!]\\
\equiv\ &\mathit{fromInteger}\ n
\end{aligned}$$

and simiarly for $\mathit{fromRational}$. Morphism check complete.

10.3.3 Adding variables

Our expressions so far denote numbers. Now let's add variables. We will use just integer-valued variables, to keep the example simple.[5] The new constructor:

$$\mathit{VarInt} :: \mathit{String} \to E\ \mathit{Int}$$

This small syntactic extension requires a significant semantic extension. An expression now denotes different values in different binding environments, so the semantic domain is a function:

$$\begin{aligned}
&\textbf{type } \mathit{Env} = \mathit{TMap}\ \mathit{String}\ \mathit{Int}\\
&\textbf{type } \mathit{Model}\ a = \mathit{Env} \to a\\
&[\![\cdot]\!] :: E\ a \to \mathit{Model}\ a
\end{aligned}$$

The semantic function carries along the environment and looks up variables as needed:

$$\begin{aligned}
&\mathit{look} :: \mathit{String} \to \mathit{Model}\ \mathit{Int}\\
&\mathit{look}\ s\ e = \mathit{sample}\ e\ s\\
&[\![\mathit{Literal}\ x]\!]\ _ = x\\
&[\![\mathit{VarInt}\ v]\!]\ e = \mathit{look}\ v\ e\\
&[\![\mathit{Add}\ \ a\ b]\!]\ e = [\![a]\!]\ e + [\![b]\!]\ e\\
&[\![\mathit{Mult}\ a\ b]\!]\ e = [\![a]\!]\ e * [\![b]\!]\ e\\
&[\![\mathit{Recip}\ a]\!]\ \ e = \mathit{recip}\ [\![a]\!]\ e
\end{aligned}$$

Again, we must ask: is $[\![\cdot]\!]$ a numeric morphism? In order for the question to even make sense, the semantic domain must inhabit the numeric classes. Indeed it does, as we saw in Section 10.1.

Looking again at the numeric methods for functions, we can see a familiar pattern in the $[\![\cdot]\!]$ definition just above. Using the function methods from Section 10.1, we get a more elegant definition of $[\![\cdot]\!]$:

$$\begin{aligned}
&[\![\mathit{Literal}\ x]\!] = \mathit{pure}\ x\\
&[\![\mathit{VarInt}\ v]\!] = \mathit{look}\ v\\
&[\![\mathit{Add}\ \ a\ b]\!] = [\![a]\!] + [\![b]\!]\\
&[\![\mathit{Mult}\ a\ b]\!] = [\![a]\!] * [\![b]\!]\\
&[\![\mathit{Recip}\ a]\!]\ \ = \mathit{recip}\ [\![a]\!]
\end{aligned}$$

With this $[\![\cdot]\!]$ definition, the morphism properties follow easily.

10.3.4 Handling errors

So far, expression evaluation is assumed to succeed. The variable environment yields values for every possible variable name, which is unrealistic. Let's now add error-handling to account for unbound variables and other potential problems. The *result* of a variable look-up or an expression evaluation will be *either* an error message or a regular value.

$$\begin{aligned}
&\textbf{type } \mathit{Result}\ a = \mathit{Either}\ \mathit{String}\ a\\
&\textbf{type } \mathit{Env} = \mathit{Map}\ \mathit{String}\ \mathit{Int}\\
&\textbf{type } \mathit{Model}\ a = \mathit{Env} \to \mathit{Result}\ a\\
&\mathit{look} :: \mathit{String} \to \mathit{Model}\ \mathit{Int}\\
&\mathit{look}\ s\ e \mid \mathit{Just}\ n \leftarrow \mathit{lookup}\ e\ s = \mathit{Right}\ n\\
&\qquad\qquad \mid \mathit{otherwise} \qquad\qquad\quad = \mathit{Left}\ (\texttt{"unbound: "} +\!\!+\ s)\\
&[\![\cdot]\!] :: E\ a \to \mathit{Model}\ a
\end{aligned}$$

Again, we'll rely on the Model having numeric instances, which is the case, thanks to the definitions in Sections 10.1 and 10.2.

As with the semantic function in Section 10.3.3, the numeric methods for the semantic domain do almost all the work, leaving our semantic definition nearly intact. The only change is for literals:

$$[\![\mathit{Literal}\ x]\!] = \mathit{pure}\ (\mathit{pure}\ x)$$

As before, most of the morphism proofs are trivial, due to the simplicity of the definitions. The $\mathit{fromInteger}$ and $\mathit{fromRational}$ morphisms use $\mathit{Literal}$:

$$\begin{aligned}
&[\![\mathit{fromInteger}\ x]\!]\\
\equiv\ &\quad \{\ \mathit{fromInteger}\ \text{definition}\ \}\\
&[\![\mathit{Literal}\ (\mathit{fromInteger}\ x)]\!]\\
\equiv\ &\quad \{\ [\![\cdot]\!]\ \text{on}\ \mathit{Literal}\ \}\\
&\mathit{pure}\ (\mathit{pure}\ (\mathit{fromInteger}\ x))\\
\equiv\ &\quad \{\ \mathit{fromInteger}\ \text{on}\ \mathit{Either}\ a\ \}\\
&\mathit{pure}\ (\mathit{fromInteger}\ x)\\
\equiv\ &\quad \{\ \mathit{fromInteger}\ \text{on}\ a \to b\ \}\\
&\mathit{fromInteger}\ x
\end{aligned}$$

and similarly for $\mathit{fromRational}$.

10.3.5 Self-optimizing expressions

So far our expressions involve no optimization, but it's easy to add. For example, replace the simple method definition $(+) = \mathit{Add}$ with the following version, which does constant-folding and recognizes the additive identity:

$$\begin{aligned}
&\mathit{Literal}\ x + \mathit{Literal}\ y = \mathit{Literal}\ (x + y)\\
&\mathit{Literal}\ 0 + b \qquad\quad = b\\
&a + \mathit{Literal}\ 0 \qquad\quad = a\\
&a + b \qquad\qquad\quad = \mathit{Add}\ a\ b
\end{aligned}$$

Other methods can perform similar optimizations.

With this change, we must re-check whether $[\![a + b]\!] \equiv [\![a]\!] + [\![b]\!]$. There are four cases to consider, corresponding to the four cases in the $(+)$ definition.

$$\begin{aligned}
&[\![\mathit{Literal}\ x + \mathit{Literal}\ y]\!]\\
\equiv\ &[\![\mathit{Literal}\ (x + y)]\!]\\
\equiv\ &x + y\\
\equiv\ &[\![\mathit{Literal}\ x]\!] + [\![\mathit{Literal}\ y]\!]
\end{aligned}$$

$$\begin{aligned}
&[\![\cdot]\!]\ (\mathit{Literal}\ 0 + b)\\
\equiv\ &[\![b]\!]\\
\equiv\ &0 + [\![b]\!]\\
\equiv\ &[\![\mathit{Literal}\ 0]\!] + [\![b]\!]
\end{aligned}$$

The third case is similar; and the fourth case is as in Section 10.3.1.

Notice that the correctness of the second case hinges on the very mathematical property that it implements, i.e., that zero is a left identity for addition. Any such property can be encoded into the implementation and then verified in this way. The discipline of type class morphisms ensure that all optimizations are correct.

[5] Multi-type variable binding environments can be implemented with an existential data type and a GADT for type equality.

11. Fusing folds

Folds are common in functional programming, to express a cumulative iteration over a list (or other data structure), reducing to a single value. There are two flavors, left- and right-associating:

$$\mathit{foldl}\ (\odot)\ a\ [b_1, b_2, ..., b_n] \equiv (...((a \odot b_1) \odot b_2) \odot ...) \odot b_n$$
$$\mathit{foldr}\ (\odot)\ a\ [b_1, b_2, ..., b_n] \equiv b_1 \odot (b_2 \odot ...(b_n \odot a)...)$$

Written recursively,

$$\begin{array}{l}\mathit{foldl} :: (a \to b \to a) \to a \to [b] \to a \\ \mathit{foldl}\ (\odot)\ a\ [\,] \quad\ = a \\ \mathit{foldl}\ (\odot)\ a\ (b : bs) = \mathit{foldl}\ (\odot)\ (a \odot b)\ bs\end{array}$$

$$\begin{array}{l}\mathit{foldr} :: (b \to a \to a) \to a \to [b] \to a \\ \mathit{foldr}\ \_\ a\ [\,] \quad\ = a \\ \mathit{foldr}\ (\odot)\ a\ (b : bs) = b \odot (\mathit{foldr}\ (\odot)\ a\ bs)\end{array}$$

There is also a strict variant of *foldl*, which eliminates a common form of space leak.

If a fold is the *only* use of a list, then lazy evaluation and garbage collection conspire to produce efficient use of memory. Laziness delays computation of list cells until just before they're combined into the fold result (using $(\odot)$), after which they become inaccessible and can be collected. This pattern allows list generation and folding to run in constant space, regardless of the length of intermediate list. (Compile-time fusion can make this process even more efficient.) Non-strict and right folds can leak due to thunks.

On the other hand, if a single list is folded over twice, then the entire list will be kept in memory to make the second fold possible. Borrowing a common example, consider the following naïve implementation of the average of the values in a list:

$$\begin{array}{l}\mathit{naiveMean} :: \mathit{Fractional}\ a \Rightarrow [a] \to a \\ \mathit{naiveMean}\ xs = \mathit{sum}\ xs\ /\ \mathit{fromIntegral}\ (\mathit{length}\ xs)\end{array}$$

Each of *sum* and *length* can be defined as a (strict) left-fold:

$$\begin{array}{l}\mathit{sum} \quad = \mathit{foldl}\ (+)\ 0 \\ \mathit{length} = \mathit{foldl}\ (\lambda a\ \_ \to a + 1)\ 0\end{array}$$

Max Rabkin (2008) gave an elegant way to fuse these two *foldl* passes into one, reducing the space usage from linear to constant. We'll see that Max's optimization and its correctness flow from a semantic model and its type class morphisms.

As a first step, let's look at a simpler setting: turning a *pair* of non-strict left foldings into a single one. Given combiners $(\odot)$ and $(\ominus)$ and initial values e and e', come up with $(\boxdot)$ and e'' such that one fold replaces a pair of them:

$$(\mathit{foldl}\ (\odot)\ e\ bs, \mathit{foldl}\ (\ominus)\ e'\ bs) \equiv \mathit{foldl}\ (\boxdot)\ e''\ bs$$

11.1 Folds as data

We want to combine the two left folds *before* supplying the list argument, rather than after. To do so, make a data type for the partial application that does not include the final (list) argument.

$$\mathbf{data}\ \mathit{FoldL}\ b\ a = F\ (a \to b \to a)\ a$$

The *FoldL* is not just a representation; it has meaning, which is partially applied folding:

$$\begin{array}{l}[\![\cdot]\!] :: \mathit{FoldL}\ b\ a \to ([b] \to a) \\ [\![\mathit{FoldL}\ (\odot)\ e]\!] = \mathit{foldl}\ (\odot)\ e\end{array}$$

With this representation and meaning, we can rephrase our goal: given two left folds f and f', find another fold f'' such that

$$([\![f]\!]\ bs, [\![f']\!]\ bs) \equiv [\![f'']\!]\ bs$$

In other words,

$$[\![f]\!] \mathbin{\&\&\&} [\![f']\!] \equiv [\![f'']\!]$$

where $(\&\&\&)$ is a common operation on functions (more generally, on arrows (Hughes 2000)):

$$\begin{array}{l}(\&\&\&) :: (x \to a) \to (x \to b) \to (x \to (a, b)) \\ f \mathbin{\&\&\&} g = \lambda x \to (f\ x, g\ x)\end{array}$$

Of course, we want a systematic way to construct f'', so we're really looking for a combining operation

$$\mathit{combL} :: \mathit{FoldL}\ c\ a \to \mathit{FoldL}\ c\ b \to \mathit{FoldL}\ c\ (a, b)$$

such that for for all f and f'

$$[\![f]\!] \mathbin{\&\&\&} [\![f']\!] \equiv [\![f\ `\mathit{combL}`\ f']\!]$$

This last form is tantalizingly close to a type class morphism. What's missing is a shared type class method.

11.2 Generalized zipping

Consider the types of the following three functions:

$$\begin{array}{lll}\mathit{zip} & :: \mathit{Stream}\ a \to \mathit{Stream}\ b \to \mathit{Stream}\ (a, b) \\ (\&\&\&) & :: (c \to a) \to (c \to b) \to (c \to (a, b)) \\ \mathit{combL} & :: \mathit{FoldL}\ c\ a \to \mathit{FoldL}\ c\ b \to \mathit{FoldL}\ c\ (a, b)\end{array}$$

These functions and more can be subsumed by a more general notion of *zipping*. Streams, functions, and folds also have in common that they provide a "unit", which will turn out to be useful later. These two common features are embodied in a type class:

$$\begin{array}{l}\mathbf{class}\ \mathit{Zip}\ f\ \mathbf{where} \\ \quad \mathit{unit} :: f\ () \\ \quad (\star) \ :: f\ a \to f\ b \to f\ (a, b)\end{array}$$

Streams and functions zip. Do folds?

The *Zip* class corresponds to *Monoidal* class (McBride and Paterson 2008, Section 7), *except* that *Monoidal* requires a *Functor*, which is not the case with *FoldL*.

Generalized $(\star)$ gives us our target TCM. Zipping a left fold means finding a semantic *Zip* morphism, i.e.,

$$\begin{array}{l}[\![\mathit{unit}]\!] \quad \equiv \mathit{unit} \\ [\![f]\!] \star [\![f']\!] \equiv [\![f \star f']\!]\end{array}$$

As shown in (Elliott 2009b), the following definition suffices:[6]

$$\begin{array}{l}\mathbf{instance}\ \mathit{Zip}\ (\mathit{FoldL}\ b)\ \mathbf{where} \\ \quad \mathit{unit} = F\ \mathit{const}\ () \\ \quad F\ (\odot)\ e \star F\ (\ominus)\ e' = F\ (\boxdot)\ (e, e') \\ \quad\quad \mathbf{where} \\ \quad\quad\quad (a, a') \boxdot b = (a \odot b, a' \ominus b)\end{array}$$

11.3 Variations

Right folds work similarly. *Strict* folds are trickier but also important, to prevent space leaks. See (Rabkin 2008) and (Elliott 2008).

Can we add more instances for *FoldL*, with corresponding semantic morphisms? Let's try *Functor*:

$$\begin{array}{l}\mathbf{instance}\ \mathit{Functor}\ (\mathit{FoldL}\ b)\ \mathbf{where} \\ \quad \mathit{fmap}\ h\ (\mathit{FoldL}\ (\odot)\ e) \equiv \mathit{FoldL}\ (\ominus)\ e'\ \mathbf{where}\ ...\ ??\end{array}$$

The functor morphism law says that

$$[\![\mathit{fmap}\ h\ (\mathit{FoldL}\ (\odot)\ e)]\!] \equiv \mathit{fmap}\ h\ [\![\mathit{FoldL}\ (\odot)\ e]\!]$$

Substituting our partial *fmap* definition on the left and $(\circ)$ for *fmap* on the right, as well as the definition of $[\![\cdot]\!]$, the property becomes

$$\mathit{foldl}\ (\ominus)\ e' \equiv h \circ \mathit{foldl}\ (\odot)\ e\ \mathbf{where}\ ...$$

[6] With this definition, the $(\star)$ morphism property fails when $f \equiv \bot$ or $f' \equiv \bot$. To fix this problem, use lazy patterns in the definition. My thanks to Ryan Ingram for pointing out this bug.

At this point, I'm stuck, as I don't see a way to get *foldl* to apply h just at the end of a fold. Max solved this problem by adding a post-fold function to the *FoldL* type (Rabkin 2008). Generalizing from his solution, we can extend *any* zippable type constructor to have *Functor* instance, and many to have *Applicative*, as we'll see next.

12. Enhanced zipping

To get from *Zip* to *Functor* and *Applicative*, add a final phase

$$\mathbf{data}\ \mathit{WithCont}\ z\ c = \forall a.\, \mathit{WC}\ (z\ a)\ (a \to c)$$

The "$\forall a$" keeps the intermediate value's type from complicating the type of a *WithCont*.

Given a semantic function $\langle\!\langle \cdot \rangle\!\rangle :: z\ a \to u\ a$, for a functor u, the meaning of *WithCont* is

$$\begin{array}{l}[\![\cdot]\!] :: \mathit{WithCont}\ z\ c \to u\ c \\ [\![\mathit{WC}\ z\ k]\!] = \mathit{fmap}\ k\ \langle\!\langle z \rangle\!\rangle\end{array}$$

For instance, adding Max's post-fold step just requires wrapping *FoldL* with *WithCont*:

$$\mathbf{type}\ \mathit{FoldLC}\ b = \mathit{WithCont}\ (\mathit{FoldL}\ b)$$

Since the meaning of a *FoldL* is a function, the *fmap* in $[\![\cdot]\!]$ is function composition, exactly as we want for applying a post-fold step. It's now easy to define instances.

12.1 Functor and Zip

The *Functor* instance accumulates final steps:

$$\begin{array}{l}\mathbf{instance}\ \mathit{Functor}\ (\mathit{WithCont}\ z)\ \mathbf{where} \\ \quad \mathit{fmap}\ g\ (\mathit{WC}\ z\ k) = \mathit{WC}\ z\ (g \circ k)\end{array}$$

The *Zip* instance delegates to the inner *Zip*, and pairs up final transformations.

$$\begin{array}{l}\mathbf{instance}\ \mathit{Zip}\ z \Rightarrow \mathit{Zip}\ (\mathit{WithCont}\ z)\ \mathbf{where} \\ \quad \mathit{unit} = \mathit{WC}\ \bot\ (\mathit{const}\ ()) \\ \quad \mathit{WC}\ \mathit{fa}\ \mathit{ka} \star \mathit{WC}\ \mathit{fb}\ \mathit{kb} = \mathit{WC}\ (\mathit{fa} \star \mathit{fb})\ (\mathit{ka} \times \mathit{kb})\end{array}$$

The semantic function is a *Functor* morphism and is sometimes a *Zip* morphism, including the case of folds (Elliott 2009b).

12.2 Applicative

As pointed out in (McBride and Paterson 2008, Section 7), the *Applicative* class has an equivalent but more symmetric variation, which is the (strong) lax monoidal functor, which has the interface of *Zip*, in addition to *Functor*. Using the standard *Applicative* instance for lax monoidal functors:

$$\begin{array}{l}\mathbf{instance}\ (\mathit{Functor}\ z, \mathit{Zip}\ z) \Rightarrow \\ \qquad \mathit{Applicative}\ (\mathit{WithCont}\ z)\ \mathbf{where} \\ \quad \mathit{pure}\ a\ \ = \mathit{fmap}\ (\mathit{const}\ a)\ \mathit{unit} \\ \quad \mathit{wf} \circledast \mathit{wx} = \mathit{fmap}\ \mathit{app}\ (\mathit{wf} \star \mathit{wx}) \\ \mathit{app}\ (f, x) = f\ x\end{array}$$

the morphism proof runs into a bit of trouble for *pure*:

$$\begin{array}{ll}[\![\mathit{pure}\ a]\!] & \equiv [\![\mathit{fmap}\ (\mathit{const}\ a)\ \mathit{unit}]\!] \\ & \equiv \mathit{fmap}\ (\mathit{const}\ a)\ [\![\mathit{unit}]\!] \\ & \equiv \mathit{fmap}\ (\mathit{const}\ a)\ \mathit{unit} \\ & \equiv \mathit{pure}\ a \quad \text{-- note: only sometimes}\end{array}$$

This property is satisfied for some functors (including $((\to)\ a)$) but not others (e.g., $[\,]$), so the *unit* morphism property holds only for *some* functors z (with semantic model u), including the meaning of *FoldL*. Continuing,

$$\begin{array}{ll}[\![\mathit{fs} \circledast \mathit{xs}]\!] & \equiv [\![\mathit{fmap}\ \mathit{app}\ (\mathit{fs} \star \mathit{xs})]\!] \\ & \equiv \mathit{fmap}\ \mathit{app}\ [\![\mathit{fs} \star \mathit{xs}]\!] \\ & \equiv \mathit{fmap}\ \mathit{app}\ ([\![\mathit{fs}]\!] \star [\![\mathit{xs}]\!]) \\ & \equiv [\![\mathit{fs}]\!] \circledast [\![\mathit{xs}]\!]\end{array}$$

Thanks to this morphism, we have a zipping law for folds:

$$\begin{array}{l}\mathit{liftA}_2\ h\ (k \circ \mathit{foldl}\ (\odot)\ e)\ (k' \circ \mathit{foldl}\ (\ominus)\ e') \\ \equiv \\ [\![\mathit{liftA}_2\ h\ (\mathit{WC}\ k\ (\mathit{FoldL}\ (\odot)\ e))\ (\mathit{WC}\ k'\ (\mathit{FoldL}\ (\ominus)\ e'))]\!]\end{array}$$

and similarly for liftA_3, etc. The proof is a direct application of the semantics and the morphism properties. This transformation could be automated with rewrite rules (Peyton Jones et al. 2001).

Returning to the example of combining two folds into one.

$$\begin{array}{l}\mathit{naiveMean} :: \mathit{Fractional}\ a \Rightarrow [a] \to a \\ \mathit{naiveMean}\ \mathit{xs} = \mathit{sum}\ \mathit{xs}\ /\ \mathit{fromIntegral}\ (\mathit{length}\ \mathit{xs})\end{array}$$

In point-free form,

$$\mathit{naiveMean} = \mathit{liftA}_2\ (/)\ \mathit{sum}\ (\mathit{fromIntegral} \circ \mathit{length})$$

So the zipping law applies (with $k \equiv id$).

13. Some more examples

13.1 Functional reactive programming

My first exposure to type class morphisms was in giving a denotational semantics to functional reactive programming (FRP) structured into type classes (Elliott 2009c). "Classic FRP" is based on a model of behaviors as functions of time (Elliott and Hudak 1997).

$$[\![\cdot]\!] :: \mathit{Behavior}\ a \to (\mathit{Time} \to a)$$

For instance, a 3D animation has type $\mathit{Behavior}\ \mathit{Geometry}$, and its meaning is $\mathit{Time} \to \mathit{Geometry}$.

Much of FRP can be re-packaged via type classes. In most cases, the semantics is specified simply and fully by saying that the semantic function is a TCM. For events, where the TCM principle is not applied successfully, the semantics is problematic. (For instance, the monad associativity law can fail.)

13.2 Functional image synthesis

Pan is a system for image synthesis and manipulation with a purely functional model and a high-performance implementation (Elliott 2003; Elliott et al. 2003). The semantic model for images in Pan is simply functions of continuous infinite 2D space:

$$[\![\cdot]\!] :: \mathit{Image}\ a \to (\mathbb{R}^2 \to a)$$

With this semantic model, the *Image* type has instances for *Functor* and *Applicative*, which form the basis of all point-wise operations, including cropping and blending. With the TCM principle clearly in mind, the *Image* interface can be made more convenient, regular and powerful, via instances of *Monoid*, *Monad* and *Comonad*, as well all of the number classes.

The implementation uses a self-optimizing syntactic representation, packaged as numeric class instances, akin to that in Section 10.3.5. The semantic function, therefore, is a morphism over all of the classes mentioned.

13.3 Automatic differentiation

Automatic differentiation (AD) is a precise and efficient method for computing derivatives. It has a simple specification in terms of type class morphisms, from which an elegant, correct implementation can be derived. The specification and implementations can then be generalized considerably to computing higher-order derivatives (infinitely many of them, lazily) and derivatives of functions over arbitrary vector spaces (Elliott 2009a).

14. Conclusions—advice on software design

With the examples in this paper, I am conveying some advice to my fellow programmers: when designing software, in addition to innovating in your implementations, relate them to precise and familiar semantic models. Implementing those models faithfully guarantees that your library's users will get simple semantics and expected properties, in addition to whatever performance benefits you provide.

By defining a type's denotation clearly, library designers can ensure there are no abstraction leaks. When the semantic function maps two program values to the same mathematical value, the designer must ensure that those program values be indistinguishable, in order to prevent semantic abstraction leaks. The discipline of denotational semantics, applied to data types, ensures that this condition is satisfied, simply because each operation's meaning is defined purely in terms of the meaning of its arguments.

Type classes introduce a new source of potential abstraction leaks, by allowing a single name to be used at different types. Although a type class dictates a set of necessary properties, it also leaves room for variation in semantics and implementation. For each program type, the library designer must take care that each interface (type class) implementation behaves consistently with the type's model, i.e.,

> *The instance's meaning follows the meaning's instance.*

which is to say that the semantic function is to be a "type class morphism" (TCM) for each type class implemented. When a library not only type-checks, but also *morphism-checks*, it is free of abstraction leak, and so the library's users can safely treat a program value as *being* its meaning. As a bonus, type class laws are guaranteed to hold, because they carry over directly from the semantic model. As much as good models tend to be widely reusable, effort invested in studying them is leveraged.

Finally, I see denotational, TCM-based design as in the spirit of John Reynolds's use of category theory for language design.

> Our intention is not to use any deep theorems of category theory, but merely to employ the basic concepts of this field as organizing principles. This might appear as a desire to be concise at the expense of being esoteric. But in designing a programming language, the central problem is to organize a variety of concepts in a way which exhibits uniformity and generality. Substantial leverage can be gained in attacking this problem if these concepts are defined concisely within a framework which has already proven its ability to impose uniformity and generality upon a wide variety of mathematics (Reynolds 1980).

15. Acknowledgments

This paper has benefited by comments from Richard Smith, Tim Newsham, Ganesh Sittampalam, Matt Hellige, Bas van Dijk, Jeremy Voorhis, Patai Gergely, Phil Wadler, Ivan Tomac, Noam Lewis, Nicolas Wu, Darryl McAdams, and Ilan Godik.

References

Stephen Adams. Efficient sets: a balancing act. *Journal of Functional Programming*, 3(4):553–562, October 1993.

Lennart Augustsson, Howard Mansell, and Ganesh Sittampalam. Paradise: a two-stage dsl embedded in haskell. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 225–228, 2008.

Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT sysposium on Principles of programming languages*, pages 1–13. ACM Press, 2005.

Conal Elliott. Functional images. In *The Fun of Programming*, "Cornerstones of Computing" series. Palgrave, March 2003.

Conal Elliott. More beautiful fold zipping. Blog post, November 2008. URL `http://conal.net/blog/posts/more-beautiful-fold-zipping`.

Conal Elliott. Beautiful differentiation. In *International Conference on Functional Programming (ICFP)*, 2009a. URL `http://conal.net/papers/beautiful-differentiation`.

Conal Elliott. Denotational design with type class morphisms (extended version). Technical Report 2009-01, LambdaPix, March 2009b. URL `http://conal.net/papers/type-class-morphisms`.

Conal Elliott. Push-pull functional reactive programming. In *Haskell Symposium*, 2009c.

Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, 1997.

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

Andy Gill et al. `Data.Monoid`, 2001. Source code.

Ralf Hinze. Generalizing generalized tries. *Journal of Functional Programming*, 10(4):327–351, July 2000.

John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1-3):67–111, 2000.

Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *FPCA '93: Conference on Functional Programming and Computer Architecture*, pages 52–61, New York, N.Y., 1993. ACM Press.

Saunders Mac Lane. *Categories for the Working Mathematician (Graduate Texts in Mathematics)*. Springer, September 1998.

Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, 2008.

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

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, 2006.

Max Rabkin. Beautiful folding. Blog post, November 2008. URL `http://squing.blogspot.com/2008/11/beautiful-folding.html`.

John C. Reynolds. Using category theory to design implicit conversions and generic operators. In *Semantics-Directed Compiler Generation, Proceedings of a Workshop*, pages 211–258. Springer-Verlag, 1980.

Dana Scott and Christopher Strachey. Toward a mathematical semantics for computer languages. Programming Research Group Tech. Monograph PRG-6, Oxford Univ. Computing Lab., 1971.

Joseph Stoy. *Denotational semantics: The Scott-Strachey approach to programming language theory*. MIT Press, 1977.

Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *The 16th Annual ACM Symposium on Principles of Programming Languages*. ACM, January 1989.