An Intuition For Decorated Traversable Monads

Lawrence Dunn[*]

Assumed background: Other than some familiarity with the use of monads in functional programming, not much background is necessary to solve the exercises below, although some jargon is provided for readers who like that sort of thing. A reader unfamiliar with the use of monads in functional programming may want to read this blog post in lieu of this document: `http://blog.sigfpe.com/2006/08/you-could-have-invented-monads-and.html`

Consider the grammar of the simply-typed lambda calculus, or STLC:

$$t = \texttt{var}\ x \quad | \quad \texttt{ap}\ t_1\ t_2 \quad | \quad \texttt{lam}\ (x : \tau)\ t$$

This grammar forms perhaps the simplest non-trivial example of what we have been calling a decorated traversable monad (DTM). By one definition, DTMs are monads that are additionally equipped with two extra structures:

1. A ‘decoration,’ the unofficial name we have given to a particular kind of coaction of endofunctors (which is honestly much simpler than it sounds)
2. A traversal in the sense of [1]

We additionally impose certain equations relating these structures to the underlying monad. The resulting abstraction seems adequate to develop an appreciable amount of syntax theory in general, untangled from the particulars of any one definite syntax. Roughly, the monad structure provides the ability to substitute terms for variables, the traversal an ability to iterate over the set of variable occurrences, and the decoration the ability to query, for each variable, what binding context it occurs in. The purpose of the following exercises is to describe how one might uncover the operations and coherence laws of DTMs by considering the syntax of STLC. (Actually, this document only defines decorated *foldable* monads, a less powerful but simpler structure, since this is easier to define.)

[*] Advised by Val Tannen and Steve Zdancewic. Mistakes are mine.

The setup

Throughout this document, we use `monospace font` to represent inductively defined types and their constructors. To describe the DTM structure of STLC's syntax we have to formalize it as an inductive type, but before that we must define type expressions and variables.[1]

Let `typ` be some Coq type whose terms represent the simple type expressions of STLC (simple types are base types A or arrow types $A \to B$). The particulars of `typ` are really unimportant, but for completeness we give a possible definition assuming a set of base types has been provided. We call this `typ` to avoid confusion with Coq's universe `Type`.

$$\frac{A : \texttt{base\_typ}}{\texttt{base}\ A : \texttt{typ}} \qquad \frac{\tau_1 : \texttt{typ} \qquad \tau_2 : \texttt{typ}}{\texttt{arr}\ \tau_1\ \tau_2 : \texttt{typ}}$$

We will implement variables as names, the way one does with pen and paper. This stands in contrast to, for instance, representing variables as natural numbers pointing to their binder (de Bruijn indices). Let V be some type with decidable equality, meaning that there is some function

$$\text{dec\_eq} : V \to V \to \texttt{bool}$$

that correctly decides whether two elements of V are equal. Terms of V are called 'names.' We write dec_eq x_1 x_2, a boolean value, as $x_1 == x_2$.

With `typ` and V defined, we can represent the grammar of STLC as an inductive type called `term`. To define the monadic and other structure of this grammar, we let `term` vary in the types of variables, with `term A` representing terms whose variables take values in type A. Terms are thought of as trees, and occurrences of variables—values x appearing in a subterm of the form `var` x—as leaves. Therefore one could also say `term` is defined generically in the type of leaves, a definition we give now.

$$\frac{x : A}{\texttt{var}\ x : \texttt{term}\ A} \qquad \frac{t_1 : \texttt{term}\ A \qquad t_2 : \texttt{term}\ A}{\texttt{ap}\ t_1\ t_2 : \texttt{term}\ A}$$

$$\frac{x : V \qquad \tau : \texttt{typ} \qquad t : \texttt{term}\ A}{\texttt{lam}\ (x : \tau)\ t : \texttt{term}\ A}$$

Notice V and `typ` always appear in the binder of a lambda abstraction, but leaves in a term of type `term` A have type A, which may or may not be equal to V. If we wanted to develop this theory in full generality we could make binders

[1] DTMs, as a model of syntax, have no notion of types, so we could just as well work with untyped lambda calculus. DTMs also don't care about how variables are represented, but we don't emphasize that here.

generic in V as well, but we don't need this much generality for these exercises. We identify the set of concrete terms of STLC with $\texttt{term}\ V$, i.e. terms whose leaves are names.

Finally, consider this definition of naïve substitution (naïve in the sense that it substitutes all occurrences of x, rather than only free ones). subst $x\ u\ t$ searches for occurrences of x appearing in t and replaces them with u. You should draw some pictures to visualize substitution as a process that visits each leaf and replaces some of them with new subtrees. For these exercises, it is often useful to draw substituted subtrees in a different color.[2]

$$\text{subst} : V \to \texttt{term}\ V \to \texttt{term}\ V \to \texttt{term}\ V$$

$$\begin{aligned}
\text{subst}\ x\ u\,(\texttt{var}\ y) &= \begin{cases} u & \text{if } x == y \\ \texttt{var}\ y & \text{else} \end{cases} \\
\text{subst}\ x\ u\,(\texttt{ap}\ t_1\ t_2) &= \texttt{ap}\ \ (\text{subst}\ x\ u\ t_1)\ \ (\text{subst}\ x\ u\ t_2) \\
\text{subst}\ x\ u\,(\texttt{lam}\ (y : \tau)\ t_1) &= \texttt{lam}\ (y : \tau)\ \ (\text{subst}\ x\ u\ t_1)
\end{aligned}$$

Our first goal is to reconstruct the definition of substitution abstractly.

Substitutions by Lifting

Instead of defining subst all at once, we can decompose this operation into two separate ones:

1. A "lifting" operation that recurses on syntax trees and applies some function to the leaves
2. A "local substitution" operation that takes a leaf and returns a term

Substitution can be defined as the lift of local substitution. The definitions are shown below. Convince yourself that this definition of substitution is precisely identical to the previous one. Then complete the exercises.

$$\begin{aligned}
\text{lift}\ f\,(\texttt{var}\ x) &= fx \\
\text{lift}\ f\,(\texttt{ap}\ t_1\ t_2) &= \texttt{ap}\ \ (\text{lift}\ f\ t_1)\ \ (\text{lift}\ f\ t_2) \\
\text{lift}\ f\,(\texttt{lam}\ (x : \tau)\ t_1) &= \texttt{lam}\ (x : \tau)\,(\text{lift}\ f\ t_1)
\end{aligned}$$

[2] This sort of drawing, or at least something similar, can be made rigorous. Since all of the structures we define are monoids in some monoidal category, the equations of DTMs can be expressed with a formal string calculus. This is actually how we first found them. `https://ncatlab.org/nlab/show/string+diagram`

$$\mathrm{subst}_{\mathrm{local}}\ x\ u\ y = \begin{cases} u & \text{if } x == y \\ \texttt{var}\ y & \text{else} \end{cases}$$

$$\mathrm{subst}\ x\ u\ t = \mathrm{lift}\,(\mathrm{subst}_{\mathrm{local}}\ x\ u)$$

1. What is the most general type of lift?
2. $\mathrm{lift}\ \ (\rule{2.5em}{0.4pt})\ \ t = t$
3. $\mathrm{lift}\ f\ \ (\texttt{var}\ x) = (\rule{2.5em}{0.4pt})$
4. $\mathrm{lift}\ g\,(\mathrm{lift}\ f\ t) = \mathrm{lift}\ \ (x \mapsto \rule{5em}{0.4pt})\ \ t$

Do you recognize these equations? A type constructor T with operations lift and `var` satisfying the above laws[3] is precisely a monad. In functional programming, for a general monad T, `var` is usually called the "return" or "unit" of T. lift is often seen taking its arguments in the opposite order and then called the "bind" of T. The 'lift' version here is more convenient for doing category theory.

As you may have heard, a monad can be defined as a monoid in the category of endofunctors. You do not have to know anything about this to keep going, but here is why that's interesting: all of the structured monads we study here are precisely monoids in some *sub*category of endofunctors. The choice of subcategory corresponds to extra structure on the monad. This naturally raises questions about what other kinds of monads arise this way. [4]

The next exercise examines decorated monads, monads with an extra "decoration" structure that we use to handle variable binding.

Decorated Lifting

In the previous section we could only define naïve substitution, which replaces all occurrences of a variable x with some expression. Now we only want to replace *free* occurrences of x. To do this, we need to keep track of the binders we encounter as we recurse on the syntax tree. We will redefine lift to maintain this information, the *binding context*, as a list. With access to the binding context, we can define local substitution to replace a variable y only if $x == y$ and the occurrence is free, meaning it does not occur in the list of bound names. We represent lists with these notations:

[3] See answers at the bottom.

[4] (For category theory enthusiasts.) The essential thing is that the subclass of functors contains the identity functor and is closed under composition, so that these form sub-monoidal-categories. The real force of working in the subcategory is that we also restrict attention to some kind of 'structure-preserving' natural transformations, which is where the coherence laws come from.

$$l = \texttt{nil} \quad | \quad x :: l$$

We introduce a helper function $\text{lift}_{\text{core}}$ that takes an additional argument k representing the "current" context, with the most recently seen bound name at the front of the list. Whenever our recursion encounters a lambda abstraction, the next recursive call is passed the result of pushing the new binder onto the context, which acts like a stack. lift can then be defined as $\text{lift}_{\text{core}}$ starting at the root of the tree and the empty context. Where k:list V, define $\text{lift}_{\text{core}} k\ f\ t$ as

$$\begin{aligned}
&\text{lift}_{\text{core}}\ \text{k}\ f\,(\texttt{var}\ x) = f(\text{k}, x)\\
&\text{lift}_{\text{core}}\ \text{k}\ f\,(\texttt{ap}\ t_1\ t_2) = \texttt{ap}\ \ (\text{lift}_{\text{core}}\ \text{k}\ f\ t_1)\ \ (\text{lift}_{\text{core}}\ \text{k}\ f\ t_2)\\
&\text{lift}_{\text{core}}\ \text{k}\ f\,(\texttt{lam}\ (x:\tau)\ t_1) = \texttt{lam}\ (x:\tau)\ \ (\text{lift}_{\text{core}}\ \ (x::k)\ \ f\ t_1)
\end{aligned}$$

Now define lifting in terms of $\text{lift}_{\text{core}}$, starting from the empty context

$$\text{lift}\ f\ t = \text{lift}_{\text{core}}\ \texttt{nil}\ f\ t$$

As an exercise, draw the tree structure of some STLC terms and visualize how lift operates. Observe that the value of k at any leaf $\texttt{var}\ x$ is precisely the set of binders that occur along the unique path from the leaf to the root. In general, the value of the binding context is particular to each leaf.

Now we define substitution of free variables. The only difference between this and naïve substitution is that now we verify that we aren't replacing a variable that appears in the binding context (and thus represents a bound occurrence). Convince yourself that this definition only replaces free occurrences. (Note, this substitution doesn't necessarily avoid capture if u has free variables. Can you fix this?)

$$\begin{aligned}
\text{freesubst}_{\text{local}}\ x\ u\ (k, y) &= \begin{cases} u & \text{if } x == y \text{ and } x \notin k \\ \texttt{var}\ y & \text{else} \end{cases}\\
\text{freesubst}\ x\ u\ t &= \text{lift}\,(\text{freesubst}_{\text{local}}\ x\ u)
\end{aligned}$$

Now let's investigate whether we can generalize the laws from the earlier section. You may be able to guess the correct answers just by sketching some trees and visualizing these operations.

5. What is the most general type of lift?

6. lift $(\underline{\hspace{2em}})\ t = t$

7. lift f $(\texttt{var}\ x) = (\underline{\hspace{2em}})$

8. lift $g\,(\text{lift}\ f\ t) = \text{lift}\ ((k, x) \mapsto \underline{\hspace{6em}})\ t$
   Hint: For this one, you may be tempted to write

   $$\text{lift}\ g\ \ (f(k, x))$$

   However, this is not correct. Why not? What's the fix?

A type constructor T with operations lift and var satisfying the above laws is what we have been calling a *decorated monad.* A monad can be decorated by any monoid. `term` is decorated by `list` V.

Such a monad is equivalent to a monoid in the category of (what we have been calling) decorated endofunctors. The previous definition can be considered a "Kleisli-style" definition, meaning the entire abstraction is given by generalizing the ordinary lift operation from the first section. An equivalent "algebraic" definition defines decorated monads as ordinary monads equipped with an operation

$$\text{decorate}\ : \texttt{term}\ A \to \texttt{term}\,(\texttt{list}\ V \times A)$$

satisfying some axioms not listed here. The name "decoration" refers to the process of visiting each leaf on t and attaching its binding context to it.

Folding

Now for a change of pace. Let us turn back to the ordinary (non-decorated) definition of lift of type

$$\text{lift} : \forall AB, (A \to \texttt{term}\ B) \to \texttt{term}\ A \to \texttt{term}\ B$$

In addition to this lifting, let's consider a (*prima facie*) new kind of operation altogether, a sort of "aggregation" operation we call foldMap[5]. Recall that a monoid is a type M with a "multiplication" operation of type $M \to M \to M$ (written infix $x \cdot y$) and a constant $1_M : M$, satisfying

$$x \cdot (y \cdot z) = (x \cdot y) \cdot z \quad \text{and} \quad 1_M \cdot x = x = x \cdot 1_M$$

Now let $f{:}A \to M$ be any function whose codomain is a monoid. Informally speaking, foldMap f t applies the map f to every leaf occurring in t,

[5] Another name could be "mapReduce," since this operation is essentially at the heart of the MapReduce paradigm. `https://en.wikipedia.org/wiki/MapReduce`

and then combines all of these values using the monoid operation. For example, if the leaves of t were numbers, we could use this operation to collect their sum, maximum, or other similar aggregates. Since we can use any monoid, foldMap represents a flexible way of computing or collecting information about the variables that occur in some syntax.

$$\begin{aligned}
&\text{foldMap } f\,(\texttt{var } x) = f x\\
&\text{foldMap } f\,(\texttt{ap } t_1\ t_2) = (\text{foldMap } t_1)\cdot(\text{foldMap } t_2)\\
&\text{foldMap } f\,(\texttt{lam } (x:\tau)\ t_1) = \text{foldMap } f\ t_1
\end{aligned}$$

One important use of foldMap is to enumerate the variables of t. This is achieved by aggregating with the operation $f{:}V \to \texttt{list } V$ that maps every element to a singleton list. The return value is the concatenation of these singletons.

$$\text{tolist } t = \text{foldMap } \ (x \mapsto x :: \texttt{nil})\, t$$

It is typically worthwhile to investigate how this sort of operation interacts with the monad structure. The exercises below ask you to determine how foldMap interacts with lift and `var`, and to consider another curious equation.

9. What is the most general type of foldMap?
10. foldMap $(\underline{\qquad})\, t = 1_M$
11. foldMap f $(\texttt{var } x) = \underline{\qquad}$
12. foldMap $g\,(\text{lift } f\ t) = \text{foldMap } \ (x : A \mapsto \underline{\qquad\qquad\qquad})\ t$

Do you see any overtones with the previous sets of equations? A monad T with operation foldMap satisfying the above laws is called a *foldable monad*. . . almost. Actually, we must stipulate a law

$$\text{foldMap } \ (\phi \circ f)\, t = \phi\,(\text{foldMap } f\ t)$$

where ϕ is any monoid homomorphism and $(g \circ f)(x) = g(f(x))$. The first equation above is a special case of this.

A foldable monad can also be defined as a monoid in the category of foldable functors. Unlike decorated monads, foldable monads can't be defined simply by generalizing the type of lift, which is why we have to introduce a new function foldMap. Still, you may have noticed that the previous equations bear a certain resemblance to the other ones we have considered. This can be explained by considering a more potent abstraction known as *traversability.*

Traversals

The operation foldMap is closely related to the strictly more powerful notion of a "traversal." We do not write the definition of traversals here, since it requires defining applicative functors, but traversals can be thought of as a well-behaved iteration principle over the set of leaves occurring in t. The extra strength of traversals, rather than folds, is essential to fully capture reasoning about syntax, but this topic is still under investigation.[6] The required coherence between the traversal and the monad seems clear—the remaining unanswered questions of this work seem to rest on properties of traversals in general.

A traversable monad can also be defined as monoid in the category of traversable endofunctors. Similarly to decorated monads, we can define traversable monads as ordinary monads with an extra operation called ‘traverse,’ or in terms of a ‘lift’ operation with a more general type than usual. foldMap is a special case of this generalized lift operation, which explains why coherence equations for foldable monads look like the monad laws themselves.

A monad that is both traversable and decorated is called a decorated traversable monad (DTM). DTMs can be defined in terms of an operation ‘lift’ that has a very general type, or as an ordinary monad with special operations decorate and traverse satisfying several coherence laws.

Bonus round: Set membership

One essential aspect of reasoning about syntax is reasoning about occurrences of variables. To capture this, define $x \in t$ as

$$x \in t \iff x \in_{\text{list}} \ (\text{tolist } t)$$

where tolist was defined earlier with foldMap and $x \in_{\text{list}} \ l$ refers to the obvious notion of what it means to occur in a list. Namely, x occurs in some list l if and only if one of the elements of l is equal to x.

Now, in terms of first-order logic, equality $=$, and the $\in$ relation, give characterizations of the following conditions (where t:`term` A). You do not need be formal about this—what do you *expect* to be the right answer?

13. $x \in (\texttt{var}\ y) \iff$ ______

14. $x \in \text{lift } f\ t \iff \exists\ (y : A)$ such that ____________

Can you define $x \in t$ in terms of foldMap without first enumerating the elements of t? What M should you use? What monoid operations? Write the foldMap laws specialized to this case. Do you see any parallels to the laws above?

[6] Presently it is not completely clear whether ordinary traversals are in fact quite right for this purpose. Officially we have been looking at a stronger condition we have been calling ‘injective’ traversability, which is easier to reason about.

References

[1] Mauro Jaskelioff and Ondrej Rypacek. An investigation of the laws of traversals. *Electronic Proceedings in Theoretical Computer Science*, 76, 02 2012.

Answers

1. $\text{lift} : \forall AB, (A \to \texttt{term}\ B) \to \texttt{term}\ A \to \texttt{term}\ B$

2. $\text{lift}\ \texttt{var}\ t = t$

3. $\text{lift}\ f\ \ (\texttt{var}\ x) = fx$

4. $\text{lift}\ g\,(\text{lift}\ f\ t) = \text{lift}\ \ (x \mapsto \text{lift}\ g\,(fx))\ \ t = \text{lift}\ \ (\text{lift}\ g \circ f)$

5. $\text{lift} : \forall AB, (\texttt{list}\ V \times A \to \texttt{term}\ B) \to \texttt{term}\ A \to \texttt{term}\ B$

6. $\text{lift}\ \ (\texttt{var}\ \circ \pi_{\mathrm{r}})\ \ t = t$ where $\pi_{\mathrm{r}}(k, x) = x$

7. $\text{lift}\ f\ \ (\texttt{var}\ x) = f\,(\texttt{nil}, x)$

8. $\text{lift}\ \ g\,(\text{lift}\ f\ t)\ = \ \text{lift}\ \ \big((k, x) \mapsto \text{lift}\ g^{+\mathrm{k}}\ \ (f(k, x))\big)\,t$ where $g^{+\mathrm{k}}(k', v) = g(k' \mathbin{+\!+} k, v)$

   Notice we must additionally pass k to g, or otherwise g would not observe the entire binding context.

9. The type of foldMap, in pseudo-Coq, is as follows: $\forall A, M$ such that M is a monoid, $(A \to M) \to (\texttt{term}\ A \to M)$

10. $\text{foldMap}\ \ (x \mapsto 1_M)\,t = 1_M$

11. $\text{foldMap}\ f\ \ (\texttt{var}\ x) = fx$

12. $\text{foldMap}\,g\,(\text{lift}\ f\ t) = \text{foldMap}\ \ (x : A \mapsto \text{foldMap}\ g\ \ (fx))\ \ t = \text{foldMap}\ \ (\text{foldMap}\ g \circ f)$

13. $x \in (\texttt{var}\ y) \iff (x = y)$

14. $x \in \text{lift}\ f\ t \iff \exists\ (y : A)$ such that $y \in t$ and $x \in fy$

    $\in$ can be defined in terms of foldMap using $y \mapsto x = y$, interpreted as a function into monoid on propositions given by disjunction (not conjunction). Using foldMap with disjunction on propositions is equivalent to existential quantification over the occurrences of t.