London Mathematical Society Impact150 Stories 1 (2015) 9–14
doi:10.1112/i150lms/t.0002

Categorical Monads and Computer Programming

Nick Benton

Abstract

The categorical notion of monad was first introduced into computer science as a way of structuring mathematical models of programming languages. The idea has subsequently been transferred back into computing practice, influencing the design of widely-used languages and frameworks.

Expressions in conventional *imperative* programming languages bear a superficial similarity to conventional mathematical notation, but they behave quite differently. For example, if `f` and `g` are both 'functions' or 'methods' that take an integer argument and return an integer result (we say they have *type* `int -> int` and write `f : int -> int`), the two expressions

```
f(3) + g(4)      and      g(4) + f(3)
```

are not generally equivalent, as they would be if `f` and `g` were ordinary mathematical functions. The issue is *not* that integer addition in programming languages is not commutative (it almost always is), but that calling a 'function' can do many things as well as, or instead of, computing a result depending solely on the arguments. And the order and multiplicity of these 'other things' – which are called *side-effects* – affect the behaviour of one's program. The ability to perform input/output operations is one example of a side-effect. If, say, both `f` and `g` just return their argument, but also print it to the screen, then the expressions will both have value 7, but we would see "34" printed in the first case, and "43" in the second, assuming that the operands of addition are evaluated left-to-right.

Another ubiquitous side-effect is mutable state. If the definitions (in C-like syntax) were

```
int f(int x) {
 r = 2;
 return x;
}
```

and

```
int g(int y) {
 return r;
}
```

where `r` is a global variable, then if the initial value of `r` is 0, the first expression returns 5, and the second 3. (While `f(3)` always returns 3, `g(4)` returns 0 before `f` has been called and updated `r`, but 2 afterwards.)

One response to this mismatch is to ask what mathematical objects *do* accurately model particular programming language features. Another is to design programming languages that obey reasoning principles closer to those of ordinary mathematical expressions.

Following the first line of enquiry, we have seen that, in the presence of effects, 'functions' of type `int -> int` cannot be modelled simply as elements of the function space $\mathbb{Z}^{\mathbb{Z}}$. However, assuming that the only effect in our language is global state, we *can* model such functions as elements of

$$(S \times \mathbb{Z})^{S \times \mathbb{Z}}$$

Received 20 May 2015.

2010 *Mathematics Subject Classification* 68Q55, 68N18, 68N15, 18D15, 18C15.

where S is a set suitable for modelling the store. That is to say, the interpretation (or *denotation*) of an `int -> int` 'function' in the language is a real function mapping pairs of an initial store and an integer argument to pairs of a new store and an integer result. In the simple case of a fixed set V of integer-valued variables, one naturally takes S to be $\mathbb{Z}^V$, the set of functions from variables to integers. The denotations of our integer-valued expressions are then elements of $(S \times \mathbb{Z})^S$; more specifically:

$$\begin{array}{ll} [\![\texttt{f(3) + g(4)}]\!](s) = (s_2, v_1 + v_2) & [\![\texttt{g(4) + f(3)}]\!](s) = (s_2, v_1 + v_2) \\ \text{where } (s_1, v_1) = [\![\texttt{f}]\!](s, 3) & \text{where } (s_1, v_1) = [\![\texttt{g}]\!](s, 4) \\ \text{and } (s_2, v_2) = [\![\texttt{g}]\!](s_1, 4) & \text{and } (s_2, v_2) = [\![\texttt{f}]\!](s_1, 3). \end{array}$$

The *semantic brackets*, $[\![\cdot]\!]$, map language syntax to the corresponding denotation; $[\![\texttt{f}]\!]$ and $[\![\texttt{g}]\!]$ are the interpretations of `f` and `g`. Note how order of evaluation is made explicit by the threading of intermediate states through the semantics.

Following the second path leads to *functional* programming languages, in which implicit side-effects are eschewed. Functional languages are based on the *lambda calculus*, a calculus of pure functions introduced by Church [**3**] in the 1930s to study computability and the foundations of mathematics. The language of category theory is widely used in studying structures arising in programming [**1**]. A particularly important example is the interpretation of the simply-typed variant of the lambda calculus in Cartesian closed categories, i.e. categories with finite products and exponentials, with objects of the category interpreting types, and morphisms interpreting terms [**8**].†

Moggi [**13**] made the crucial observation that the semantics of languages with various kinds of side-effect, including the treatment of state sketched above, could all be factored in the same way: as a Cartesian category $\mathcal{C}$ modelling pure (non-side-effecting) *values*, equipped with a strong monad T for interpreting potentially side-effecting *computations*. A strong monad is an endofunctor $T : \mathcal{C} \to \mathcal{C}$ equipped with natural transformations $\eta_X : X \to TX$, $\mu_X : T(TX) \to TX$ and $\tau_{X,Y} : X \times TY \to T(X \times Y)$ satisfying some coherence conditions. If $[\![\texttt{X}]\!]$ models values of some type `X`, then $T([\![\texttt{X}]\!])$ models computations of type `X`. In the case of global state, for example, one might model values in the category of sets and functions and then use the monad

$$\begin{array}{l} TX = (S \times X)^S \\ T(f : X \to Y) = \lambda c : TX. \lambda s : S. \, (s_1,\ f\, x) \text{ where } (s_1, x) = c\, s \\ \eta_X(x) = \lambda s : S. \, (s, x) \\ \mu_X(c) = \lambda s : S. \, c_1\, s_1 \text{ where } (s_1, c_1) = c\, s \end{array}$$

to interpret computations. The interpretation $[\![\texttt{X->Y}]\!]$ of a function type is $(T[\![\texttt{Y}]\!])^{[\![\texttt{X}]\!]}$, i.e. maps from values to computations. In the case of state, one can see that $(S \times Y)^{S^X}$ is isomorphic to $(S \times Y)^{S \times X}$, which was used above. Moggi also proposed factoring the semantics of effectful languages through that of a computational metalanguage, a typed lambda calculus extended with a new type constructor `T`, corresponding to the functor, and new term constructors interpreted using the monad operations.

A great range of effects can be modelled this way. For exceptions, or errors, take $TX = X + E$, the disjoint union of X with a set of exception values E. For non-determinism, $TX = \mathbb{P}_{fin}X$, the finite powerset of X. For partiality due to recursion, $TX = X_\perp$, the lift of X (adding a new 'bottom' element to X in a category of certain posets (domains) and continuous maps, rather than sets). For output $TX = C^* \times X$, pairs of sequences of characters from C with elements of X. More sophisticated examples include monads for continuations (used to interpret jumps and exotic control-flow mechanisms), resumptions (for concurrency), and probabilistic computation

†Among many other examples are the understanding of datatypes as initial algebras [**4**], coalgebras for studying automata and dynamical systems [**17**], and groupoid models of dependent type theory [**6**].

(with applications including machine learning and cryptography). Moreover, the translations from impure languages into the pure metalanguage all have the same shape.

Moggi's idea has been hugely influential in research on the semantics of programming languages, but what is perhaps more notable is the way in which this piece of theory has transferred back into practice, initially in the context of functional languages [**2**]. Programming with higher-order functions and without side-effects has many advantages: programs are concise, elegant and clear, amenable to parallelization and much easier to reason about and transform than their imperative counterparts. But the point of running a program usually *is* to cause some side-effect: reading and writing to the screen, file system or network. Furthermore, using side-effects, particularly mutable state, is often desirable for performance reasons. Functional programmers struggled for many years to incorporate input/output and other effects, without destroying the good properties of their languages, until Wadler [**18**] proposed using Moggi's semantic construction (initially for mathematically *modelling* impure languages) as an actual *programming* abstraction in pure languages, of which the best-known example is Haskell [**15**].

Monads are a central abstraction in Haskell. The language provides convenient syntactic and library support for programming with monads, including parameterizing code over different monads. Many Haskell monads are defined entirely in the language, providing a reusable abstraction of common patterns in purely functional programming (for example in parsing), whilst others, including the monads for IO and mutable state, have built-in imperative implementations, safely encapsulated by the monad abstraction. Monads make side-effecting 'actions' into first-class data values that can be composed, passed around and explicitly computed *with*, rather than things that happen as an implicit side-effect *of* computation. For example, the `IO` monad has an operation `getChar : IO Char`, representing the action that *when it is performed* will read (and yield) the next character from the input. Similarly, `putChar : Char -> IO ()` is a function that takes a character, c, and returns the action that will print c to the output (and yields a value of the trivial one-element type, `()`). The infix operator `>>=` corresponds to the Kleisli composition operation of the monad: it combines little actions into bigger ones. If m : `IO a` is an action yielding a value of type `a`, and f : `a -> IO b` is a function mapping `a`'s to `b`-yielding actions, then (m `>>=` f) : `IO b` is the action that when run will first run m, performing some input/output and yielding a value x : `a`, and then run (and yield the value of) $f\,x$, performing some more input/output. For example, (`getChar >>= putChar`) : `IO ()` is an action that first reads a character from the input and then echoes that character back to the output.

A key point is that monads like `IO` are *abstract*: the type system ensures that one can only manipulate `IO` actions using the provided operations. Crucially, there is no operation of type `IO a -> a`, so no way to 'forget' the distinction between actions and values.[†] A consequence is that (modulo termination) Haskell functions still behave like mathematical ones, depending only on their arguments and satisfying simple equational laws, even when the arguments and results may themselves be actions. A complete program is an expression of type `IO ()`, and one may imagine program execution as a process of 'simplifying' such an expression, using the usual equations of the lambda calculus, to produce an action-expression in canonical form, which is subsequently run to cause actual IO effects to be performed. This model of pure computation of impure actions reconciles the mathematician's 'is' with the programmer's 'does'.

Functional languages are not quite mainstream, but are used in many companies, from small startups to large corporations (Intel, Facebook, Standard Chartered, Verizon, Ericsson, BAE,...), and in application areas including finance, hardware design, security, and cloud and

[†]It should be admitted that real Haskell implementations do include just such an unsafe operation. But ordinary programmers are told that they should never, ever use it.

web infrastructure [**5**]. Industrial proponents of functional programming find that monadic encapsulation of side-effects and powerful type systems not only remove whole classes of bugs, but also enable the building of rich, higher-order domain-specific abstractions, so a program can be close to a purely declarative, high-level, mathematical model of, for example, a financial option [**16**] or an electronic circuit [**14**]. As part of its anti-abuse infrastructure, Facebook uses a monadically-structured Haskell library, which transparently parallelizes, batches and caches the fetching of data from external sources [**10**].

Futhermore, following an evolutionary, rather than revolutionary, path, functional ideas, and monads in particular, have seen significant adoption in more traditional languages over the last decade or so. Google's influential MapReduce model for parallel processing of large datasets is based on the observation that computations expressed in terms of a restricted set of well-known functional primitives are amenable to being automatically distributed across networked clusters of machines [**9**]. Microsoft ships both a functional language, F♯, and an object-oriented language, C♯, that has numerous functional features. These include 'Language Integrated Query' (LINQ): a uniform interface to many kinds of data (including traditional databases, XML documents, and data on the web). LINQ generalizes the standard database query language, SQL, and is explicitly designed around monads, here as a common abstraction of collections and sequences [**11**]. A further development is the Reactive Extensions (Rx) library, which also presents asynchronous events, such as mouse clicks or stock quotes, as LINQ-queryable data sources. Rx is now available in many languages and greatly simplifies programming of reactive applications by, roughly, allowing one to describe the desired relationships between event streams, rather than writing imperative code to be executed when each event occurs. Rx was originally devised by formally dualising (in the categorical sense) the types of collection operations [**12**]. F♯ also supports general monadic programming in the form of its 'asynchronous expressions', a primary use for which is scheduling communicating concurrent tasks.

The monad abstraction has now been implemented in all major programming languages (often many times) and is widely deployed, from simple uses of the error monad ($TX = 1 + X$) as a better alternative to null-pointers, to the use of Rx in both the server and the client parts of Netflix's video streaming service, smoothly orchestrating concurrent streams of asynchronous events between the user interface and backend services [**7**]. Online, there are now hundreds (possibly thousands) of articles on monads, aimed at practising programmers and making many fanciful analogies, comparing monads to spacesuits, monsters, boxes, elephants, and onions, amongst other things. While the exact relationship between some of the real-world code and the original, formal mathematical constructions may be a little loose, the fact remains that some of the most important ideas in 21^{st} century programming have their roots in 1930s mathematical logic and 1960s category theory.

References

1. M. Barr and C. Wells. *Category Theory for Computing Science*. Prentice-Hall, 1990.
2. N. Benton, J. Hughes, and E. Moggi. Monads and effects. In G. Barthe, P. Dybjer, L. Pinto, and J. Saraiva, editors, *Applied Semantics: Advanced Lectures*, volume 2395 of *LNCS*. Springer, 2002.
3. A. Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, pages 346–366, 1932.
4. J. Goguen, J. Thatcher, and E. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In R. Yeh, editor, *Current Trends in Programming Methodology*, pages 80–149. Prentice-Hall, 1978.
5. Haskell.org. Haskell in industry. `https://wiki.haskell.org/Haskell_in_industry`. Accessed April 2015.
6. M. Hofmann and T. Streicher. The groupoid interpretation of type theory. In G. Sambin and J. Smith, editors, *Twenty-Five Years of Constructive Type Theory*, volume 36 of *Oxford Logic Guides*, pages 83–111. OUP, 1998.
7. J. Husain. Netflix: End to end reactive programming. In *Commercial Users of Functional Programming*, 2013. `http://cufp.org/2013/jafar-husain-netflix-end-end-reactive-programming.html`. Accessed April 2015.

8. J. Lambek and P. J. Scott. *Introduction to higher-order categorical logic*, volume 7 of *Cambridge Studies in Advanced Mathematics*. CUP, 1988.
9. R. Lämmel. Google's MapReduce programming model – revisited. *Science of Computer Programming*, 70:1–30, 2008.
10. S. Marlow, L. Brandy, J. Coens, and J. Purdy. There is no Fork: An abstraction for efficient, concurrent, and concise data access. In *International Conference on Functional Programming*. ACM, 2014.
11. E. Meijer. The world according to LINQ. *ACM Queue*, 9, 2011.
12. E. Meijer. Your mouse is a database. *Communications of the ACM*, 55(5):66–73, 2012.
13. E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.
14. R. S. Nikhil. Types, functional programming and atomic transactions in hardware design. In *Buneman Festschrift*, volume 8000 of *LNCS*, pages 418–431. Springer, 2013.
15. S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. CUP, 2003.
16. S. Peyton Jones and J.-M. Eber. How to write a financial contract. In *The Fun of Programming*. Palgrave Macmillan, 2003.
17. J. Rutten. Universal coalgebra: A theory of systems. *Theoretical Computer Science*, 249:3–80, 2000.
18. P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2(04):461–493, 1992.

N. Benton
Microsoft Research
21 Station Road
Cambridge CB1 2FB
United Kingdom

nick@microsoft.com