

MONADS AND INTENSIONALITY

Bill Wadge

Computer Science Department
University of Victoria
Victoria, British Columbia
Canada

ABSTRACT

Monads are a concept from category theory which can be used to structure functional programs or even define nonstandard interpretations of the λ -calculus. These nonstandard functional languages share many of Lucid's unusual features, such as the distinction between synchronic and general functions. In fact Lucid is exactly one of these languages, determined by a simple stream monad. We therefore propose that monadic functional programming as the appropriate meaning of "Intensional Programming", with "Indexical Programming" reserved for languages in which intensions are indexed families of extensions.

AN UNEXPECTED BOUNTY

Every year at ISLIP we have at least one soul searching discussion on the topic "what is intensional programming". Recently we added a new twist, an animated debate about "indexical" versus "intensional" programming. This new schism will undoubtedly cause even outsiders be even more baffled than ever by the slippery notion of intensionality.

At the same time, we often have an equally spirited but more accessible discussion of the old question "is Lucid a functional language?". Many functional programmers have argued that Lucid (and indexical languages in general) are not functional languages. One of the more convincing reasons was that indexical languages lacked higher order functions. Fortunately, this defect has been remedied Wadge[1991].

Nevertheless, even with higher order functions Lucid is still very different from Miranda and Haskell. We have streams rather than lazy lists, and the streams are built into the language in a rather peculiar way. And of course Miranda and Haskell have nothing resembling GLU's user defined dimensions, or the notorious "freezing" of the original Lucid. For these reasons it is more usual to classify Lucid as a dataflow language, *rather than* as a functional language. Of course, even considered as a dataflow language Lucid still appears idiosyncratic, but that's another story.

This year I am pleased to report a new development which, I claim, gives a really good answer to all these questions. And the development is due entirely to our longtime rivals, functional programmers (especially Moggi and Wadler). They have (almost unwittingly) helped us solve some of our most perplexing issues and at the same time given Lucid a claim to far more mainstream respectability than we ever expected. At

least that's the way I see the situation. They have discovered a whole family of nonstandard functional languages which share Lucid's idiosyncracies.

This new development, of course, is the discovery (by functional programmers, especially Wadler) of the significance of *monads* for programming languages.

MONADS

Monads are a concept well known in category theory, and the basic definitions can be found in almost any text on the subject (they are sometimes called "triples" or "standard constructions"). I will try to avoid all but the most basic categorical notions and for the time being stay rather informal. Phil Wadler has written several excellent expositions of the subject, and I recommend in particular his POPL 92 tutorial Wadler[1992]. My presentation of monads is slightly different, and is intended for people who already know about indexical programming.

A monad can be thought of as an operation $(*)$ on data types or domains which, given a domain D , yields a new domain D^* . The idea is that the elements of D^* are in some sense generated by the elements of D .

The canonical example (for us indexical programmers) is to take D^* to be the collection of *streams* over D . We will keep returning to this 'standard' example because it can be very useful for understanding some of the otherwise rather abstract ideas. Of course, we can generalize streams by taking D^* to be the set of all I -families of elements of D for some index set I , but in terms of monads this doesn't take us very far. The importance of the monad approach is that it is far more general than streams or even arbitrary I -families. For that reason it is not always safe to think of elements of D^* as being simply families of elements of D . In my experience, however, it is reasonable to imagine that an element of D^* is in some sense an 'aggregate' of elements of D .

There is, of course, more to a monad than just an operation on domains. The $*$ operation also applies to maps: if f is a function from domain D to domain E , then f^* maps D^* to E^* . We can imagine that f^* works by applying f to all the components of its argument, transforming an aggregate built from D 's into an analogous aggregate built from E 's. In the case of the stream monad, f^* is the operation which applies f pointwise to a stream.

The next component of a monad is a family of mappings which map each D into D^* . We can overload our notation by letting d^* be the element of D^* which corresponds in this sense to the element d of D . The idea here is that d^* is some canonical aggregate all of whose components are d . In the case of streams, d^* is the infamous "constant stream" which has value d at every stream index. This family of mappings ensure that in some sense each D is a subdomain of D^* .

The most helpful comment I have seen describes a monad as a kind of closure operator. What this means is that iterating $*$ gives us nothing new. This doesn't mean that D^{**} is literally the same as D^* ; but it means that for each D there is a 'standard' mapping \downarrow_D from D^{**} to D^* . This family of mappings is the third component of a monad.

If we think of elements of D^* as being aggregates of D 's, then elements of D^{**} are double aggregates of D 's. From this point of view, \downarrow_D represents a standard way of collapsing a double aggregate into a single aggregate.

In general, there is no rule about how this collapse takes place. In the case of the stream monad, however, the answer should be clear. The domain D^{**} is the collection of streams of streams of elements of D . How do we collapse a stream of streams? We look at it as a doubly indexed stream and take the *diagonal*. This is hardly a new idea: it corresponds to the *latest*⁻¹ of the old Lucid, with freezing. The surprising development is that this rather bizarre operation (called *contemp* in the Lucid book) should have such a place of honour in the Holy Trinity of the stream monad. Yet we will soon see that it has earned its place.

THE MONAD LAWS

Not every 'closure' operator $*$ defines a monad. The operator $*$ and the associated collapse operator must obey certain simple rules which ensure that the nonstandard lambda calculus they define still has some of the most basic properties of the ordinary lambda calculus.

The first set of laws ensure that $*$ acts, in a sense, "uniformly" on data types and operations. They say that (1) the star of the composition of two functions is the composition of the stars; and (2) that the star of the identity map is the identity map. These laws are implicit in the category theory formulation because they amount to saying that $*$ is a *functor*.

The next law captures the idea that d^* is the result of 'copying' d to form an aggregate all of whose components are d . Given any D and E and any f from D to E , we require

$$f(d^*) = f(d)^*$$

for any d in D . (Strictly speaking, the two occurrences of $*$ denote different maps. The first denotes the map from D to D^* , and the second denotes the map from E to E^* . But we will use the short form since no confusion arises.)

Next, we have a law which formalizes the idea that collapsing maps act uniformly on double aggregates of different base types. It says that applying a function componentwise before or after a collapse gives the same results:

$$f^*(s \downarrow) = f^{**}(s) \downarrow$$

for any s in D^{**} . (Again we cut notational corners: the first \downarrow is \downarrow_D , and the second is \downarrow_E). In the categorical formulation, this amounts to saying that \downarrow is a *natural transformation* from the functor $**$ to the functor $*$.

We also require that a constant double aggregate collapses to the appropriate constant single aggregate. This means that

$$d^{**} \downarrow = d^*$$

for any d in D .

The most important rule says in effect that \downarrow is associative (hence the name “monad”, by analogy with monoids). Given any D , there are two ways to get from D^{***} to D^* . The first way is to treat it as $(D^*)^{**}$ and collapse to $(D^*)^*$, then collapse again to D^* (apply \downarrow_{D^*} then \downarrow_D). The other way is to treat D^{***} as $(D^{**})^*$ and collapse componentwise to $(D^*)^*$ then to D^* (apply $(\downarrow_D)^*$ then \downarrow_D). These must be the same. In shorthand notation,

$$t \downarrow \downarrow = t \downarrow^* \downarrow$$

for any t in D^{***} .

This last rule may seem rather complex but it guarantees that the nonstandard functional languages retain some of the usual properties of the pure lambda calculus.

SOME SIMPLE MONADS

Some examples will make it clear that the concepts are not as difficult as they might seem.

We already mentioned the ‘canonical’ example, in which D^* is the set of streams over D , f^* is the map which applies f pointwise, d^* is the stream which is constantly d , and \downarrow is the diagonal map from double streams to single streams. Only the monad associative law gives pause for thought, but it takes only a few minutes to check out the monad equations. We find that

$$\langle \langle \langle d_{i,j,k} \rangle_i \rangle_j \rangle_k \downarrow \downarrow = \langle \langle d_{i,j,j} \rangle_i \rangle_j \downarrow = \langle d_{i,i,i} \rangle_i$$

and

$$\langle \langle \langle d_{i,j,k} \rangle_i \rangle_j \rangle_k \downarrow^* \downarrow = \langle \langle d_{i,i,k} \rangle_i \rangle_k \downarrow = \langle d_{i,i,i} \rangle_i$$

Both ways of collapsing a triple stream coincide. In fact, they both correspond to taking the diagonal of the *cube*.

The functional programming monadologists do not seem to be explicitly aware that the stream operator makes a natural monad. However, Wadler has a “state reader” monad that is formally identical to the stream monad just given. The difference is that Wadler is thinking in terms of a real state, and the state reader monad is used in conjunction with a more general state monad which captures the notion of a function with side effects.

A very different example is to take D^* to be the collection of all subsets of D . If f maps D to E then f^* is the function that applies f elementwise. In other words,

$$f^* (\{d_0, d_1, d_2, \dots\}) = \{f^*(d_0), f^*(d_1), f^*(d_2), \dots\}$$

The image d^* of d is the singleton $\{d\}$.

The domain D^{**} is the collection of all sets of sets of elements of D . This is clearly not the same as D^* but the map from D^{**} to D^* is obvious: the union map, which takes a set of sets and returns the union of all the sets in its argument. Thus the double set $\{\{0,1,2\}, \{2,3,5\}, \{3,6\}\}$ (in Z^{**}) is collapsed to $\{0,1,2,3,5,6\}$. It is not hard to check that union is associative in the sense described above: both ways of collapsing a three

level set yield the set of all integers 'appearing' somewhere in it.

A slightly more interesting variant is to take sets in which each element has a real number weight associated with it, with the weights of the elements of a given set adding up to one. The collapsing operation is similar except that in taking the unions the weights of the inner elements are multiplied by the weights of the intermediate sets to which they belong. For example,

$$\{\{a^{0.5}, b^{0.5}\}^{0.4}, \{a^{0.1}, c^{0.9}\}^{0.6}\}$$

is collapsed to

$$\{a^{0.26}, b^{0.20}, c^{0.54}\}$$

Another idea is to take D^* to be elements of D 'tagged' with a natural number; in other words, D^* is the product of D and the set of natural numbers. If f maps D to E , the map f^* applies f to the data part of its argument and leaves the tag unchanged. The map from D to D^* takes d to d^0 .

An element of D^{**} is an element of D with two possibly different tags. The collapsing map replaces the two tags by the largest of them (so that, eg, $(d^{12})^{18}$ is collapsed to d^{18}). The associativity of the maximum operator guarantees the associativity of this collapsing operation.

These are only a few of the many possible examples. It should be clear elements of D^* may not be anything like streams of elements of D .

ISWIM

Monads were originally 'discovered' (for computer science) by Moggi[1989], who saw in them a way to structure some otherwise cumbersome denotational language semantics. About the same time Michael Spivey [1990] proposed them as the basis of a 'functional' approach to exceptions and nondeterministic choice. Wadler [1990] proposed a generalization of list comprehension indexed by a monad, but in Wadler [1992] seems to be suggesting programming in nonstandard, 'monadized' functional languages.

Monads have proved useful because they allow us to define, in a uniform way, nonstandard interpretations of the lambda calculus; or, more conveniently, of Landin's Iswim.

Iswim is a sugared version of the lambda calculus: in addition to ordinary application and lambda abstract, it has block-like construct for defining local variables. Landin originally used his "where" clause (adopted by Lucid) but for our purposes we will take the simpler "let" construct.

Iswim is a form of the *applied* lambda calculus - it has primitive operations with a predefined interpretation (for example, arithmetic operations over the natural numbers). In typed Iswim, there are various ground types which are interpreted as different domains. Given any types τ and σ , we can also form the product type $\tau \times \sigma$ and the function type $\tau \rightarrow \sigma$.

We will not discuss recursion because it slightly complicates the semantics (everything has to be a complete partial order) without really shedding extra light on the basic concepts.

A particular member of the Iswim family is determined by an interpretation I . An interpretation assigns a domain to each ground type. It also specifies a collection of typed constants and assigns to each of them an object of the appropriate functionality. For example, if $I(\mathbf{t})$ is the integers, I might assign to the symbol $+$ of type $\mathbf{t} \times \mathbf{t} \rightarrow \mathbf{t}$ the usual addition function over the integers.

The semantics of Iswim is simplicity itself, and is based on the notion of referential transparency. The meaning of an application is the result of applying the meaning of the expression being applied to the meaning of the argument.

To give the semantics of blocks and λ expressions (which bind variables) we must introduce the notion of an environment - a function which assigns to each variable a value of the appropriate functionality.

The meaning of the lambda expression $\lambda x.B$ in the environment M is the function f where for any value v appropriate for x , $f(v)$ is the value of the body B in the environment M' identical to M except that M' assigns v to x . In more concise notation,

$$\llbracket \lambda x.B \rrbracket_M = \lambda v \llbracket B \rrbracket_{M[x/v]}$$

Similarly, the meaning of $\text{let } x \text{ be } A \text{ in } B$ is the meaning of B in which x had been reassigned the value of A . More concisely,

$$\llbracket \text{let } x \text{ be } A \text{ in } B \rrbracket_M = \llbracket B \rrbracket_{M[x/\llbracket A \rrbracket_M]}$$

MONADIZING ISWIM

We can now, given a monad $*$ and an interpretation I , define a nonstandard version of $\text{Iswim}(I)$ in which expressions which appear to denote elements of a domain D actually denote elements of D^* . The notion of aggregate which $*$ defines is available but not explicitly, as another data type. Instead, it is built right into the language.

Only the meanings of the ground types are altered; the type constructors are interpreted in the usual way. Thus if $I(\mathbf{t})$ is Z , $I(+)$ will have functionality $Z^* \times Z^* \rightarrow Z^*$.

For example, the set monad gives us nondeterministic Iswim; the weighted set monad gives us probabilistic Iswim; the tagged Iswim gives us a real-time Iswim; and the stream monad gives us Luswim (not Lucid; see next section).

The first step is to monadize the interpretation I , giving an interpretation I^* . For a basic type τ , $I^*(\tau) = I(\tau)^*$. For example, terms in monadized Iswim which are of type integer really denote integer aggregates, not simple integers.

The treatment of basic operations is a little more complex.

If c is a nullary operation, ie a constant, $I(c)$ is a simple element d of the appropriate domain D . Then we set $I^*(c) = d^*$. For example, the numeral 5 will now denote not the number 5 but the canonical aggregate all of whose components are 5.

If f is a first order unary function symbol (of type $\sigma \rightarrow \tau$ for ground types σ and τ), we let $I^*(f) = I(f)^*$. For example, if sq is the square function, $I^*(\text{sq})$ is the function which squares the components of an aggregate.

The problems begin with binary operations and with higher order constants. Suppose that ∇ is a primitive of type $\sigma \times \tau \rightarrow \theta$. This means that $I(\nabla)$ has functionality $I(\sigma) \times I(\tau) \rightarrow I(\theta)$, and that $I^*(\nabla)$ should have functionality $I(\sigma)^* \times I(\tau)^* \rightarrow I(\theta)^*$. What should $I^*(\nabla)$ be? It seems that we cannot proceed without further assumptions.

One possibility is to require a kind of outer product operation, which takes a pair of aggregates and give us a double aggregate of pairs. In all the examples, the product is obvious: with streams, we get the double stream of pairs, and with sets, we get a set of sets of pairs, organized by their first component.

Given such a product, we can define $I(\nabla)$ as follows. Let a and b be to aggregates in $I(\sigma)^*$ and $I(\tau)^*$, respectively. We form the product of a and b and then collapse it, and apply $I(\nabla)^*$ to the result (ie apply the original map $I(\nabla)$ pointwise).

It is easy to check that with the familiar examples this gives the expected result. For example, it means that the sum of two streams is the stream of sums of corresponding pairs (the usual Lucid definition). For sets, it means that the sum of two sets is the set of all possible pairs of elements chosen from the respective sets. For tagged integers, it means that the sum of two tagged integers is their sum tagged with the maximum of the tags of the operands.

Clearly we can continue in this way extend I^* to handle all first order functions. So we have already generalized Lucid.

The simplest monadized language is $\text{Iswim}(I^*)$ but it is really of very little interest because the primitives are all just $*$ 'd versions of ordinary primitives. We can make it more interesting by extending I^* to an interpretation J which gives meanings to new primitives, ones which were not available in $\text{Iswim}(I)$. For example, with the stream monad we can add the Lucid operators; with the set monad, a function amb which takes the union of its arguments; or with the tagged monad, a primitive race which returns the argument with smallest tag.

LIFTING HIGHER ORDER PRIMITIVES

Now we turn to the problem of handling higher-order primitives. We seem to need even more assumptions, and the most obvious one to make is that we are dealing with a cartesian closed category. In lay terms this means, roughly speaking, that we have a function domain operation. This is a very strong assumption but most of the categories we deal with have this property.

Once we make this assumption, we see that the problem of $*$ -ing higher order primitives is already implicit in problem of $*$ -ing first order binary operations. Suppose that h is a map with functionality $D \times E \rightarrow F$. The map h determines a map g of functionality $D \rightarrow (E \rightarrow F)$, and the problem we solved in the last section is essentially that of finding a corresponding map h' of functionality $D^* \rightarrow (E^* \rightarrow F^*)$.

Let us begin by considering the pointwise map h^* of functionality $D^* \rightarrow (E \rightarrow F)^*$. This is not quite what we want, because the codomain is $(D \rightarrow E)^*$, not $D^* \rightarrow E^*$. The former are *aggregates of functions*, whereas the latter are *aggregate functions* (functions mapping aggregates to aggregates).

In the context of streams, the difference is familiar: it is the difference between function streams and stream functions, as explained for example in Wadge & Ashcroft [1985, pp231-232]. We see there that every function stream determines a filter (stream function) in a simple way; and that the filters obtained this way are what we call *synchronic*.

We cannot duplicate the construction given there in this more abstract context, but there is an alternate plan which works. Consider an element a of $(D \rightarrow E)^*$, and let d be an element of D . Since a is an aggregate of functions, we can feed d to each of the components of a , and obtain an aggregate of results (each of which will be in E). This implies that a determines an element k of $D \rightarrow E^*$. Furthermore, k^* will be in $D^* \rightarrow E^{**}$ and if we compose k^* with \downarrow_E , we get an element of $D^* \rightarrow E^*$.

To summarize: there is a map from $(D \rightarrow E)^*$ to $D^* \rightarrow E^*$, so that *every function aggregate determines an aggregate function*. Furthermore, those functions obtained in this way are of a particularly simple kind; they are ‘really’ functions from $D \rightarrow E^*$. This formulation accurately generalizes the situation in indexical programming. It even generalizes the notion of *synchronic* function. A map f from D^* to E^* is generalized synchronic iff for some k in $D \rightarrow E^*$,

$$f(s) = k^*(s) \downarrow_E$$

for every s in D^* .

Once we have this result, it is straightforward to ‘lift’ higher order maps like h (above). In particular, the desired map h' is simply the result of composing h^* with the conversion map from $(D \rightarrow E)^*$ to $D^* \rightarrow E^*$.

CALL BY VALUE VERSUS CALL BY NAME

The new languages described above are not really nonstandard versions of Iswim. Each such language is just $\text{Iswim}(J)$ where J is an extension of the interpretation I^* . The new language is still a member of the Iswim family, and abstraction, application, and `let` are still interpreted in the normal way.

The language $\text{Iswim}(J)$ corresponds to the language Luswim defined in Wadge & Ashcroft [1985,p66]. User defined functions are not necessarily synchronic; they are passed entire aggregates as arguments, and return entire aggregates as results.

Sometimes, however, this generality is more than we need. Consider, for example, the set monad, so that $\text{Iswim}(J)$ can be thought of as nondeterministic lambda calculus. The expression $\lambda x \ x * x$ represents the intensional function which takes a nondeterministic argument and multiplies it by itself.

This sounds like the square function $\lambda x \ x ** 2$ but they are not the same. The square function, given a multi-valued integer (eg $\{1,3,5\}$) returns a multi-valued integer whose

values (in this case $\{1,9,25\}$) are just the squares of the values of the argument. More precisely, if sq is the ordinary square function from $Z \rightarrow Z$, $\lambda x \ x^{**2}$ denotes, in the extended language, the componentwise function sq^* from Z^* to Z^* .

On the other hand, in the intensional language the expression $\lambda x \ x \times x$ denotes a function which takes a multi-integer and returns a multi-integer whose values are all possible products pairs of values of its argument. This function, given $\{1,3,5\}$, returns $\{1,3,9,15,25\}$. It is not an extensional function in disguise: not of the form f^* for any f from Z to Z .

We could explain the situation this way: functions in this language are given the entire multi-integer as an argument, and in the course of computing the function any reference to the actual parameter involves an independent sampling. Sometimes, however, we would like a different calling convention: in which the arguments are sampled *once*, at calling time, and references in the body all refer to the same sampled value.

The difference here is basically the old Algol distinction between call by name and call by value, but in a far more general context.

This led Wadler and others to suggest an alternate call by value interpretation of lambda which is guaranteed to define synchronic functions. The basic idea of the alternate interpretation is that the bound variable would range over extensions (elements of D), rather than intensions (elements of D^*).

It is not hard to find the right definition. Given an environment M , and a lambda expression $\lambda x.B$, we first define the function f by

$$f(d) = \llbracket B \rrbracket_{M[x/d^*]}$$

The functionality of f is $D \rightarrow E^*$ but, as we have already seen, we can convert it a generalized synchronic function in $D^* \rightarrow E^*$, which we take as the meaning of the expression. More concisely,

$$\llbracket \lambda x.B \rrbracket_M = \lambda v \ (\lambda d \ \llbracket B \rrbracket_{M[x/d^*]})^*(v) \downarrow_E$$

Similarly, we can define the call by value let as follows:

$$\llbracket \text{let } x \text{ be } A \text{ in } B \rrbracket_M = (\lambda d \ \llbracket B \rrbracket_{M[x/d^*]})^*(\llbracket A \rrbracket_M) \downarrow_E$$

The monadic distinction between call by name and call by value specializes, in the case of the stream monad, to concepts very familiar to indexical programmers. (Notice the prominence of \downarrow , a.k.a. *latest*⁻¹ in the semantics of the call by value let).

Even the earliest versions of Lucid (Ashcroft & Wadge [1978]) recognized the special role of synchronic functions (they were called *mappings*). This version of the language provided two constructs for defining functions (*function* and *mapping* clauses) which correspond exactly to the call by name and call by value interpretations of λ given above. This version also provided two block constructs (*produce* and *compute* clauses) which correspond exactly the call by name and call by value versions of *let*.

Later versions had only one function construct but retained the and combined the two kinds of *let* into the single *freezing* construct which proved quite unpopular in some

circles.

Clearly, we could also define a monadized version of Iswim which supported both kinds of λ and both kinds of `let`. We could even follow the Lucid lead and combine the two `let`'s for a kind of generalized freezing. The details are not particularly difficult, although we must be careful about higher order functions.

CONCLUSIONS

The difference between the two λ 's is this: in one the variable range over intensions (streams), while in the other they range over extensions (data objects). This strongly suggests that these terms can be carried over to the monad generalization. Elements of D are *extensions*, while elements of D^* are *intensions*.

If we accept this generalization, we can settle the question of the relationship between *intensional* and *indexical*. My suggestion is this: by *intensional programming* we mean programming in any of these nonstandard monadic languages. We reserve *indexical* for the special case in which D^* is the set of all families of elements of D over some *index* set. This would please many logicians who have never been convinced that possible world semantics constitutes a complete explanation of intensionality.

My next suggestion is that the nonstandard Iswim's be admitted to the illustrious family of *functional* languages, even if this does amount to a generalization of the concept. This suggestion has been implicitly accepted by Wadler and others, who regard monads as being an important advance in functional programming (and rightly so). Of course, in accepting this suggestion they must acknowledge that Lucid is, indeed, a functional language.

My final suggestion is that we indexical programmers can stop worrying that our languages are idiosyncratic or, to use a famous phrase, "messy and weird". Lucid shares the following weird characteristics with all the monadic languages:

- (1) The notion of aggregate is built right into the language; it is not just another data type.
- (2) The distinction between extensional and intensional is fundamental. But there are only simple intensions, not double or triple or multiply nested intensions.
- (3) There are three classes of functions, namely pointwise, synchronic, and general. Basic data functions are all pointwise. All pointwise functions are synchronic.
- (4) Variable binding constructs can exist in two forms, extensional and intensional, depending on whether the values of the variables range over extensions or intensions.
- (5) The extensional version of λ defines functions guaranteed to be synchronic.

My suggestions, then, is that Intensional Programming is far more general than Indexical Programming. The latter is just one of the possible interpretations of the former. In fact Indexical Programming is just one *extension* of Intensional Programming.

REFERENCES

- E. Ashcroft and W. Wadge [1978]. *Scope structures and defined functions in Lucid*, POPL 78, pp17-22, ACM
- E. Moggi [1989]. *Computational lambda-calculus and monads*, Symposium on Logic and Computer Science, Asilomar, California, IEEE.
- M. Spivey [1990]. *A functional theory of exceptions*, Science of Computer Programming, 14(1):25-42, June 1990.
- W. Wadge and E. Ashcroft [1985]. *Lucid, the dataflow programming language*, (310pp), Academic Press.
- W. Wadge [1991] *Higher order Lucid*, ISLIP 91, pp62-69, SRI International.
- P. Wadler [1990]. *Comprehending Monads*, Conference on Lisp and Functional Programming, Nice, France, ACM.
- P. Wadler [1992]. *The essence of functional programming*, POPL 92, pp1-14, ACM.