Recursion Diagrams: ideas for a Geometry of Formal Methods

Dr. Andrew Butterfield,
Foundations and Methods Group,
Trinity College,
Dublin University,
Andrew.Butterfield@cs.tcd.ie

October 23, 1998

Abstract

This paper describes work leading towards the concept of a *Geometry of Formal Methods*[Mac96],[HM97], which explores the relationship between various formal description techniques and aspects of modern abstract algebraic theories with a strong geometric interpretation, in particular such concepts as fibre-bundles, sheaves and related ideas in topology and category theory. Inspired by ideas and notions of seeking a geometry of computing and of formal methods, and with the category theoretic concepts of topos in mind, we explore how such a geometry might be expressed in concrete diagrams, and explore their ability to lay clear some of the concepts behind tail recursion optimisation. We also indicate how this approach can be used in an exposition of various published program transformation rules in this area. We also show the use of category theoretic notions to help explain the similarity of two apparently quite different diagrams. All of this points towards a future foundation for our geometry, both in diagrammatic and algebraic form, in the area of category theory.

1 Introduction

This paper describes work leading towards the concept of a *Geometry of Formal Methods*, as originally proposed in [Mac96]. The idea there was to find an analogue, in the area of formal methods, of the *Cartesian Duality* which was so successful in uniting Euclidean geometry with algebra. These ideas were expanded upon in [HM97], which explored the relationship between various formal description techniques and aspects of modern abstract algebraic theories with a strong geometric interpretation, in particular such concepts as fibre-bundles, sheaves and related ideas in topology and category theory. In this paper we adopt a different tack, taking as a cue various remarks made regarding the concept of an algorithm [Mac98] as being in some sense a trajectory through some form of space. The notion of algorithm or program state space is not new [Dij76, p10] [DS89, pp4–5], but here we attempt to use diagrams to visualise the behaviour of different algorithms, particularly those that implement the same

function. A key goal here is to explore the notion of a geometry of algorithms as an aid to *seeing* the relationships between those of interest. The connection with formal methods is retained in that our main interest is using diagrams to aid in understanding various correctness-preserving transformation rules.

As a concrete area to work in, and because it has been of interest to our colleagues in the Foundations and Methods Group, we choose to use the relationship between tail and other forms of recursion as a vehicle to explore this aspect of formal methods "geometry".

The rest of this paper is organised as follows: first we introduce the concepts and notation, by considering the example of the recursive factorial algorithms, looking at tail and linear (or "naive" [Mac90]) variants; then we look at some more complex examples, before moving on to consider the use of the proposed diagrams as a means of describing and comprehending various recursion transformation rules; finally, we relate this material back to contemporary related work, such as the ideas of [HM97], automata theory, and various approaches possible from within category theory, and explore suggested future avenues of research.

2 Setting the Scene: Factorial

Our starting point is an observation about the motion of a tail-recursive function in some sort of "trajectory space", as made by my colleague, Dr. Mícheál Mac an Airchinnigh.

Consider the tail-recursive form of factorial:

$$f(n) \triangleq f[n]1 \tag{1}$$
$$f[0]a \triangleq a \tag{2}$$
$$f[n]a \triangleq f[n-1](n \cdot a) \tag{3}$$

Notation Note : We make overloaded use of the symbol f here, using it to denote both the top-level call, as well as the tail-recursive form. Here the two uses are clearly distinguished on account of their different signatures. In order to emphasis this point, and the fact that we prefer to express multivariate functions as higher-order single variable forms, using currying, we put the first curried argument in square brackets. Hence $f(3)$ denotes a call at the top level to $f : \mathbb{N} \to \mathbb{N}$, whereas $f[3]$ denotes a call to the curried tail recursive form: $f : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$.

When we look at the recursive step, we see that it can be viewed as the repeated action of a parameterised morphism ($f[\_]$), whose parameter decreases as it operates upon an accumulator argument:

$$a_0 \xrightarrow{f[3]} 3a_0 \xrightarrow{f[2]} 6a_0 \xrightarrow{f[1]} 6a_0 \xrightarrow{f[0]} 6a_0$$

Diagram 1

We can also view $f[n]$ itself as a mapping taking $a \mapsto n! \cdot a$, and hence as an instance of an affine transformation — this connection into a possible geometric

interpretation was mentioned in [Mac96]. However, our concern here is more with the diagrammatic way in which it was presented above.

Now let us consider the linear (or "naive") recursive version:

$$f(0) \triangleq 1 \tag{4}$$
$$f(n) \triangleq n \cdot f(n-1) \tag{5}$$

An attempt to display a trajectory similar to that above fails almost immediately, because the multiplication here is "external", occurring after the recursive call has returned, and is a binary operation, which requires two arguments, which makes it harder to come up with a nice linear picture, similar to that above for tail recursion.

In order to simplify matters, we will recast all binary operators into curried form, so that $a \oplus b$, for an arbitrary operator $\oplus$, becomes $\oplus[a]b$ instead. We see that this reformulation allows us to express things in a much neater diagrammatic form than if product constructions were present. As our binary or 2-place operators are all generally total and well-behaved, we do not need to concern ourselves with subtleties regarding the relationship between $\_ \oplus \_ : A \times A \to A$ and $\oplus[\_]\_ : A \to A \to A$, such as their domains of definedness.

The key to a successful display of recursion using diagrams comes from having an explicit representation of the control argument for the recursion — this is the argument which is tested to see if a base case has been reached, and which is decremented according to some well-founded ordering for every recursive call. The diagram above (Diagram 1) shows the data (accumulator argument) manipulation explicitly, but the control argument changes are implicit, visible only in the changing parameterisation of the morphism $f[\_]$.

Before we look at tail recursion, let us introduce our diagrams and associated conventions,by considering the standard linear recursive form of factorial, recast using currying for binary operators:

$$f(0) \triangleq 1 \tag{6}$$
$$f(n) \triangleq \times[n](f(\mathrm{dec}\, n)) \tag{7}$$

First, let us consider the function as a whole — f has two formal placeholders, one for its input argument, called n, and another implicit one for result, which we call r. We indicate these placeholders by stacking them one above each other as follows:

n

r

Because these placeholders are associated with a given call to f, we may stress their linkage by connecting them with lines as shown below (this comes in useful when we look at binary recursion later on):

However, for single recursion, they usually clutter up the diagram, so we tend to omit them when the connection they denote is obvious from context.

We now consider the base case, when $n = 0$. In this case, the value 1 is returned, but we wish to express this showing 0 "becoming" 1 in some sense. We achieve this by using a constant function — we adopt the combinator notation of [CF58], which denotes the constant function that always returns x by Kx. Armed with this, we produce the diagram for the base case:

At this stage we should point out that we are handling conditionals implicitly, producing separate diagrams for each case, so strictly speaking we should label this diagram with a guard:

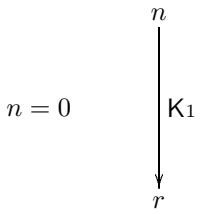
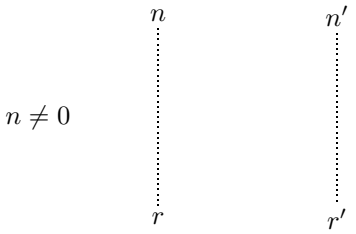


Diagram 2

We now look at the recursive case. We have two calls involved, the current one, and the recursive sub-call that is made in order to complete the calculation. We adopt the convention of putting the current call placeholders on the left, and the sub-call's on the right:



For the sub-call we label the placeholders with the same names but decorated with primes. Note that we have no indication yet of the roles of the placeholders, to wit., if they stand for inputs or outputs of the function. These roles are made clear in the final diagram.

We use labelled arrows to express the transformations that occur

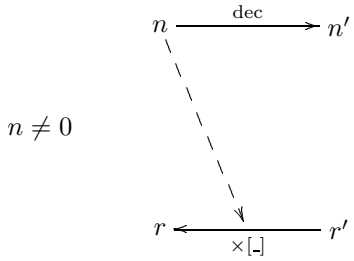


Diagram 3

We see that $n' = \mathrm{dec}\, n$ and that $r = \times[n]r'$ (in that order). The arrows clearly indicate which placeholders are inputs and which are outputs. The diagonal arrow, feeding n to the first curried argument of $\times[\_]$ is shown dashed because it is technically of a different type to those connecting n, n' and r, r'.

We can also see that the effect of the recursive step is to make the sub-call with work pending, namely the multiplication of the eventual result by the control input argument. It is this need to "come back", to the current value of n, and the pending multiplication which requires stack storage to keep track of it.

The overall flow becomes clear when we construct a diagram showing the complete execution of $f(3)$, were we have omitted the diagonal cross links for clarity :

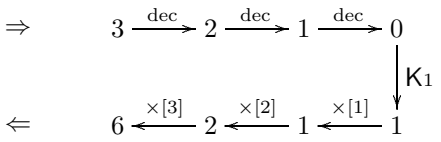


Diagram 4

Here you clearly see the stack being built on the way out, as it is the sequence of reversed arrows on the bottom. These same right to left arrow show the stack unwind sequence as the successive calls terminate, rippling back up. We have used double arrows to show the flow of arguments into the function ($\Rightarrow$)and the result flow out ($\Leftarrow$).

The conceptual clarity afforded by these diagrams becomes much more obvious when we show the tail recursive form in a similar manner. We shall find that the optimisation of replacing tail recursion by iteration, becomes immediately obvious.

The tail recursive form using curried functions and ignoring the initial call $(f[n]1)$ is:

$$f[0]a \triangleq a \tag{8}$$

$$f[n]a \triangleq f[\mathrm{dec}\, n](\times[n]a) \tag{9}$$

The base case is presented diagrammatically as:

$$n$$

$$n = 0$$

$$a \qquad \Rightarrow$$

Diagram 5

where again, the double arrow indicates that the value a is the final output of the function. We can also express this more compactly, omitting the guard and showing $n = 0$ explicitly:

$$0$$

$$a \qquad \Rightarrow$$

Here we have departed in a sense from our convention, because we have only two place holders, for n and a, but no explicit result place holder. However we make two points in our defence — one now, the other deferred until we have completed our look at tail recursive factorial. The first point is that the accumulator argument a effectively is the result placeholder in this case, although in general tail recursion it would be modified by the base case, before the function terminated. An example of this is given later when a transformation rule is discussed.

The recursive step is as follows:

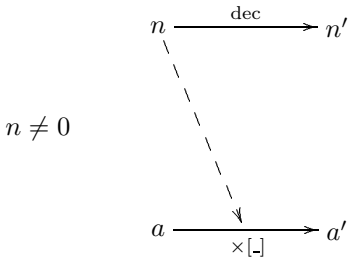
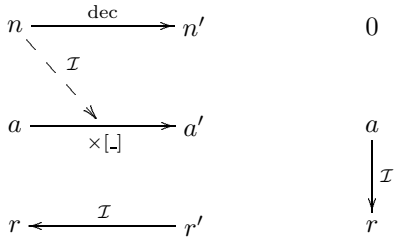


Diagram 6

Immediately we can see that there is no work to be done once the sub-call is made, so there is no need to keep the current value of n around, or to stack up any pending $\times[\_]$ operations. Also we are now well-placed to make our second point as to why a result placeholder (r) is not present. If it were, the recursive

and base step diagrams would be:



Here we have that same out-and-back flow we saw for the linear recursion, but it is not immediately obvious that the back flow is vacuous and redundant, consisting solely of identity functions ($\mathcal{I}$). However, it is a good way to illustrate the inefficiency of *unoptimised* tail recursion[1]. In general, to avoid excessive clutter, we omit explicit use of $\mathcal{I}$, and adopt the convention that an undecorated arrow denotes an application of an identity function.

Returning to our tail recursive factorial, the following diagram shows the computation of $f[3]1$, showing clearly the one-way trajectory from input to result :

$$\Rightarrow \qquad 3 \xrightarrow{\text{dec}} 2 \xrightarrow{\text{dec}} 1 \xrightarrow{\text{dec}} 0$$

$$\Rightarrow \qquad 1 \xrightarrow{\times[3]} 3 \xrightarrow{\times[2]} 6 \xrightarrow{\times[1]} 6 \qquad \Rightarrow$$

Diagram 7

The key point here is that these diagrams make the relationship between linear and tail recursive factorial much easier to see, as well as the opportunity for optimising tail calls. We see this as being analogous to using geometric diagrams to illustrate a proof of some geometric property, even if the proof itself is done algebraically, via cartesian or other such coordinates. Part of our goal is to develop techniques and diagrams that are pedagogically sound ways of conveying these often difficult concepts. The clarity of these diagrams in displaying the "out-and-back" nature of naive recursion in contrast to the "go-straight-to-the exit" track of optimised tail recursion, is for us a good validation of the potential of this diagrammatic approach to be an effective device for communicating such ideas.

2.1 A Key Observation: the Wreath Product

Up to this point, we have presented this diagram notation as an aid to understanding, *to seeing*, the relationship between related algorithms. But these diagrams also triggered other more technical associations, by virtue of their similarity to other diagrams. A case in point is the very strong resemblance of the tail-recursive step diagram above to that used to illustrate the concept of *Wreath Product*, used in automata theory [Eil74, pp26–32].

[1] Tail recursion is optimised by converting it into iteration

The Wreath Product of total functions $\lambda\colon P \to P'$ and $\delta\colon P \to Q \to Q'$, denoted here by $\lambda \circledcirc \delta$ is defined as

$$\lambda \circledcirc \delta \quad : \quad P \times Q \to P' \times Q' \tag{10}$$

$$(\lambda \circledcirc \delta)(p, q) \quad \triangleq \quad (\lambda(p), \delta[p]q) \tag{11}$$

We can see its behaviour in the following diagram

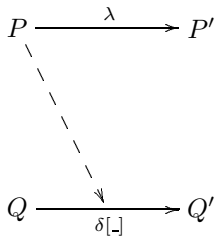


Diagram 8

Here we use P and Q to denote the domain and range carrier sets of the λ and δ operators, rather than instances of the set elements being operated upon, as has been the case in previous diagrams in this paper.

We can immediately see that the tail recursive step of factorial can be expressed, with a minor change of signature[2], as:

$$f(n, a) = f((\text{dec} \circledcirc \times)(n, a)) \tag{12}$$

Simply put, we specify a tail recursive step by simply providing the function that reduces the control argument, and the function that computes the new accumulator value from the current input values. The wreath product provides the "plumbing".

What is the significance of all this? First, note that the wreath product is used to model finite state machines, as distinct from pushdown stack automata. The key point about transforming linear recursion to a tail form is to be able to eliminate the need for a stack — to convert an algorithm that requires a pushdown automata into one that needs a simple finite state machine. We find that our diagrams end up showing the tail recursive step as having the same form as an abstraction (the wreath product) used to model finite state machines. We consider this a strong validation of the approach being adopted here. If we are to achieve our goal of finding a geometry of computing, or of formal methods, then the more we can exhibit close ties with our approach between various related areas (such as finite state machine theory, and tail-call optimisation), the more confidence we may have that we are on the right track.

3 More Complicated Examples

We now proceed to look at some more complicated examples, in order to get a better feel for how well these diagrams work. We first look at an example of binary recursion, the well known Fibonacci function. There has been considerable

[2]As already stated, we consider $f(a, b)$ and $f[a]b$ as the same entity, for the purposes of this paper.

work reported on in the literature on transforming various forms of recursion into tail recursive form [BD77, AK82, BW82, PP86, Par90]. We look at a few examples of such transformation rules, here taken from [Par90, §6.5].

3.1 Fibonnacci Function

Many forms of binary recursion can be transformed into linear (naive) and hence tail recursive forms. The classic example of this is the Fibonacci function which is usually presented with the following binary recursive definition:

$$f(0) \triangleq 0 \tag{13}$$

$$f(1) \triangleq 1 \tag{14}$$

$$f(n+2) \triangleq f(n+1) + f(n) \tag{15}$$

for which we can produce the following diagram for the recursive case:

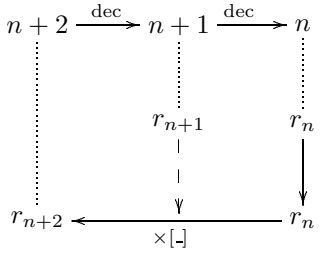
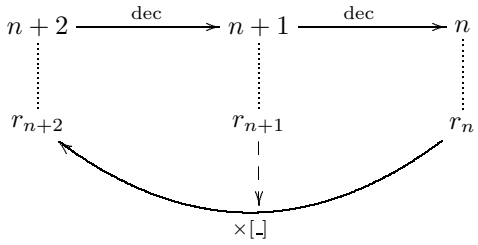


Diagram 9

Note that here we need to indicate explicitly which placeholders are linked in a given call, because of the presence of two recursive sub-calls. We have also made our placeholder labelling a bit more self-explanatory.

The transformation of the above into tail recursion usually involves a so-called "eureka" step, the details of which will follow shortly. However we can use this diagrammatic method to split the eureka step into two — a conceptually simpler and more obvious eureka step and a straightforward interpretation of the resulting diagram.

We note that both $f(n+2)$ and $f(n+1)$ will invoke $f(n)$, and decide that it would be nice if we could implement everything as a linear sequence of calls. So, we simply re-arrange the above diagram so that the placeholders for each call are in a straight line, and make the appropriate re-routing of all the arrows. We ignore the fact that some arrows seem to jump from one call to another, *without passing through appropriate placeholders*. We ignore this complication, and so obtain the following diagram:

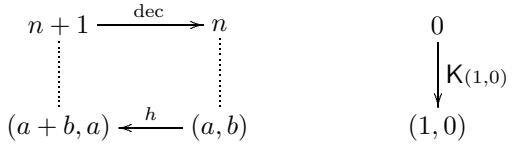


Note that this diagram is in some sense "illegal", because of the "jumping" arrow — the curved arrow at the bottom. We resolve this by introducing new (result) placeholders at the points where the jumping arrows cross, and then combine these with the existing result placeholders into result placeholder pairs.

What we have done is derived in a graphical manner the classic "eureka" idea for linearising Fibonnaci, which is to generalise the function to return both $f(n)$ and $f(n-1)$ together — we call this function g, noting that $(f(n+2), f(n+1)) = (f(n+1) + f(n), f(n+1))$, and so obtain :

$$g(0) \triangleq (1,0) \tag{16}$$
$$g(n+1) \triangleq h(g(n)) \tag{17}$$
$$\textbf{where} \quad h(a,b) = (a+b, a) \tag{18}$$

This gives us the following simple diagram :



The key feature to note here is the absence of effective "wreathing" (the accumulator computation ignores the control argument) — the control variable (n) serves solely to count the application of h to the initial argument, so we can reverse the bottom arrows immediately to obtain:

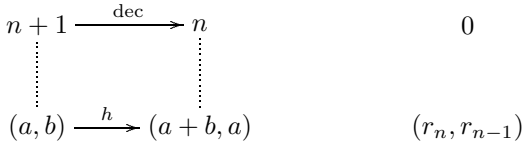


Diagram 10

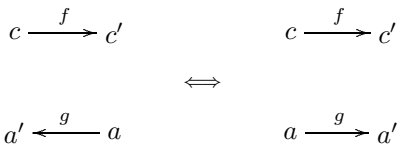
from which we instantly obtain the tail recursive form of g :

$$g(n) \triangleq \pi_2(g[n](1,0)) \tag{19}$$
$$g[0](a,b) \triangleq (a,b) \tag{20}$$
$$g[n+1](a,b) \triangleq g[n](a+b, a) \tag{21}$$

and the iterative form : $f(n) = \pi_2(h^n(1,0)), n > 0$.

It is quite clear that the diagrams provide a good aid to seeing and understanding the relationship between the various algorithms.

The rule applied above is an instance of the following *No-Wreath Swap Rule*, which is valid in the absence of any wreathing (as indicated by the absent diagonal arrow):



4 General Recursion Transformations

We now look at some program transformation rules, using as our example those found in Partsch's 1990 book, [Par90]. In particular, our focus of interest is on the contents of section §6.5, pp296–310, entitled "Simplification of Recursion". We have rewritten the material in our style, but keep the same names as used there, in order to facilitate comparisons.

The first rule is *Simplification of Linear Recursion I*. Consider:

$$f(x) \triangleq B(x) \;\rightarrow\; H(x)\ ,\ p[K(x)]f(K'(x)) \tag{22}$$

subject to the condition that p is associative ($p[a](p[b]c) = p[p[a]b]c$), with neutral element E ($p[E]a = p[a]E = a$).

We can immediately draw the following diagram illustrating both the recursive step (left side) and the base case (right side):

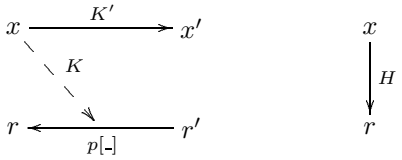
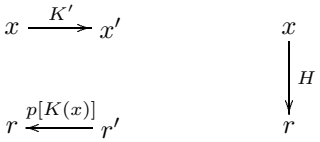
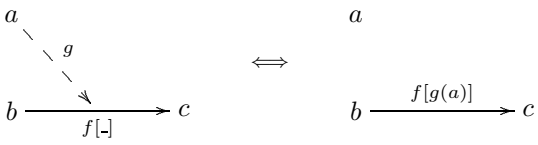


Diagram 11

For clarity we have omitted mention of B here. A simplified version of this diagram omits the explicit dashed diagonal arrow linking $K(x)$ to the first argument position of $p[\_]$:



Here we are using the following *Dashed Eliding Rule* which allows us to hide diagonals, to reduce clutter:



Note however, the diagonal is still present, if implicit, and that the No-Wreath Swap rule may not be subsequently applied.

According to *Simplification of Linear Recursion I* we can re-define f as the following tail recursive form

$$f(x) \triangleq f[x]E \tag{23}$$

$$f[x]y \triangleq B(x) \rightarrow p[y]H(x)\ ,\ f[K'(x)](p[y]K(x)) \tag{24}$$

This leads to the following diagram, which is complicated by the necessity to keep p's arguments in the correct order, as in general it is not commutative:

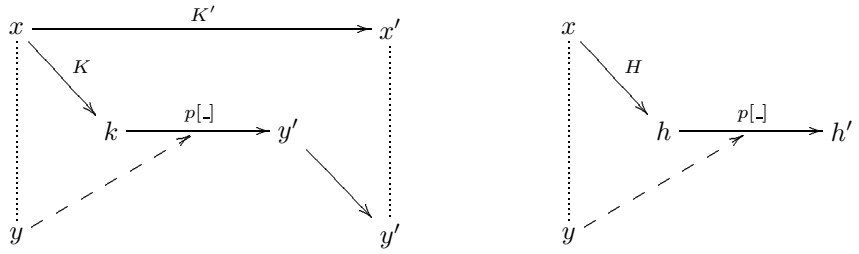
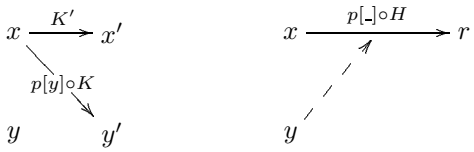


Diagram 12

Here k is a placeholder for $K(x)$ and $h = H(x)$. This diagram is quite complex, but we can remove some of the clutter, using the Dashed Eliding Rule, as follows:



Here there is no explicit arrow linking y to the first argument position of $p[\_]$, in the recursive case.

Let us consider an example where $f(x_0)$ is such that $B((K')^3(x_0)) = \mathsf{true}$. The linear recursive execution has the following diagram:

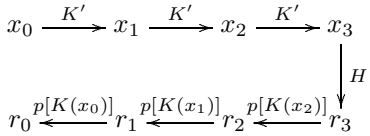


Diagram 13

From this, by tracing through, we can see that what is returned is the expression:

$$r = p[\kappa^0 x](p[\kappa^1 x](p[\kappa^2 x](\vartheta^3 x)))$$

where $\kappa^n = K \circ (K')^n$ and $\vartheta^n = H \circ (K')^n$.

The tail recursive execution is indicated as :

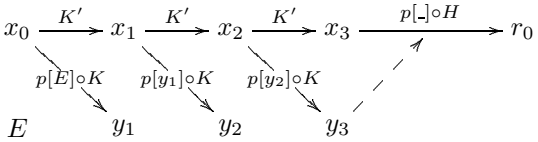


Diagram 14

Here the expression returned is

$$r = p[p[p[E](\kappa^0 x)](\kappa^1 x)](\kappa^2 x)](\vartheta^3 x)$$

By comparison we can clearly see the need for associativity and neutrality in order to make this work, but also the lack of a need for commutativity.

This rule is a key motivation behind the choice of defining recursive functions on sequences using $\langle x \rangle \frown \sigma$, rather than $x : \sigma$, that is adopted, for instance, in [Mac90]. The structure $(\Sigma^\star, \frown, \Lambda)$ is a monoid so $p[x]y = x \frown y$ and $E = \Lambda$ have the relevant associativity and neutrality properties. The cons operator is not a binary operator at all, and has a signature incompatible with that of $p[\_]$ However an analogous result is obtainable if we consider functions from "cons-lists" $(x : \sigma)$ to "snoc-lists" $(\sigma \,;\, x)$, as has been considered in [Wad87].

The next example from [Par90] is *Recursion simplification II*. Given

$$f(x) \triangleq B(x) \;\rightarrow\; H(x) \;,\; p[f(K(x))]E(x) \tag{25}$$

subject to the condition that K is invertible, in the sense that $K(x)$ being defined means that $K^{-1}(K(x)) = x$, we can redefine f as

$$f(x) \;\triangleq\; f_x[f'(x)]H(f'(x)) \tag{26}$$

$$f'(x) \;\triangleq\; B(x) \;\rightarrow\; x \;,\; f'(K(x)) \tag{27}$$

$$f_x[y]z \;\triangleq\; y = x \;\rightarrow\; z \;,\; f_x[K^{-1}(y)](p[z](E(K^{-1}(y)))) \tag{28}$$

Notation Note : Here we have an argument (x) to f which is unchanged throughout. To emphasise its once-off parametric effect, we denote it as a subscript (f_x).

The rationale behind this transformation seems very obscure. The key is to recognise that f' computes the value that x will have when recursion terminates, so that $f[\_]$ is called with this final value, and effectively uses K^{-1} to "recurse backwards". Let us use our diagrams to illustrate this transformation.

First, we show the linear recursive form :

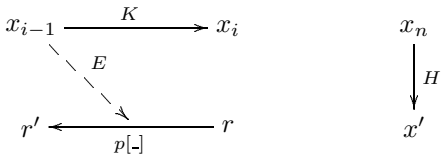
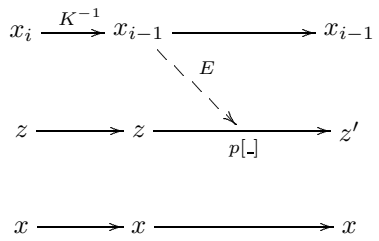


Diagram 15

Now we show the tail recursive version, in two steps. First f' simply goes for the terminating x value:

$$x_0 \xrightarrow{K} x_1 \cdots x_{n_1} \xrightarrow{K} x_n | B(x_n)$$

Then $f[\_]$ uses this value to compute backwards :



We could simplify the diagram a bit more, as follows :

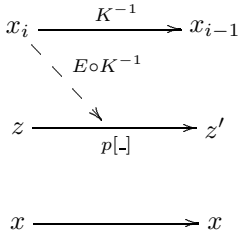
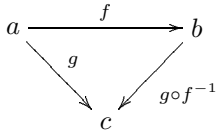
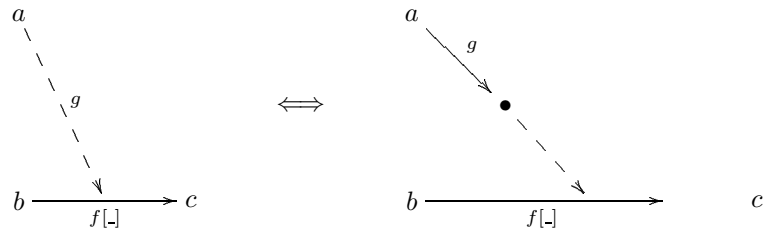


Diagram 16

Comparison of this step with the linear recursive step above shows one to be the converse of the other. To recognise this clearly we can state as a lemma that the following diagram commutes :

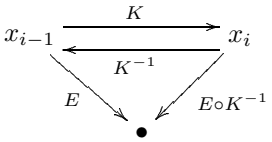


We then apply the following *Dashed Composition Rule*:



to both the linear recursive and tail forms, discarding all but the "triangle" portion to obtain

Merging the two diagrams and comparing with our lemma gives our desired result:



In this case, if $f'(x)$ is known in advance (typically when $B(x)$ is of the form $x = x_0$, hence $x_0 = f'(x)$, for any x), then the rule can be simplified, as stated in [Par90, p300]. We simply observe that we shorten the first part of the diagram, and replace it by x_0.

5 Related and Future work — towards a foundation

The purpose in looking for a geometric view for computing, is not solely to allow interesting pictures to be drawn. In continuous mathematics, great strides have been made since Descartes first connected geometry to algebra, with a large amount of abstract mathematics developed, specifically in areas such as topology, fibre bundles and sheaf theory, as previously hinted in [HM97]. With this in mind, we also seek a firm mathematical foundation for our geometric view — we hope to find it in the broad area of category theory [Wal91, LS97], with particular emphasis on topoi [Hug98]. The desire to find a categorical foundation for the diagrams shown here has influenced the way they were constructed and the (often implicit) rules governing them. However these rules are not yet established, because the most suitable categories have yet to be identified. Initial work exploring categories made it very clear, early on, that we needed a much better feel for what the diagrams would express, before sensible categorical choices could be made. In computing and category theory, there is already some material where certain categorical constructs are interpreted as diagrams

of different aspects of computing systems. An excellent introduction to this area is [Wal91], which has, among other things: a description of products in a suitable category with boolean functions, being used to represent digital logic circuits [Wal91, pp32–39]; and sums in another category with arithmetic and test functions being used to represent program flowcharts [Wal91, 45–52]. An important point to note here is that combining these aspects require categories with both products and sums, as well as various other constructions. A good class of candidate categories with these properties are topoi, hence our interest in them.

A comprehensive review of categories as a basis for an algebra of functional programs can be found in [Bd97]. The structures adopted there (called "allegories") seem different in nature to topoi, and are oriented towards allowing functional programs to be specified as relations, and towards providing a means of transforming such relations into corresponding functions. However as stated in [Bd97, p109], their concept of a "tabular allegory with a unit and power object" is a topos. This is certainly an area we should investigate, in order to see how our diagrams fit with this approach.

In the area of the π-calculus and similar formalisms, work has been done on graphical calculi ([Mil94],[Par95]). However these differ markedly from the work reported here in a major sense. Those graphical calculi are ways of representing the structure of process algebra terms in a visual manner, as well as giving a graphical description of the process of "reduction". In [CL95] we see a graphical calculus for representing and manipulating formulas of the relational calculus.

In contrast, the graphs in this paper portray the state of execution of a (functional) program in terms of variables and changes to their states. There is no sense in which diagrams have a direct interpretation as terms of some lambda-calculus like language, or any of the transformation rules an interpretation as reduction or conversion.

6 Summary

Inspired by ideas and notions of seeking a geometry of computing and of formal methods, and with the category theoretic concepts of topos in mind, we have explored how such a geometry might be expressed in concrete diagrams, and explored their ability to lay clear some of the concepts behind tail recursion. We have seen how the relationship between linear and tail recursion in the factorial function can be visualised, and explored the simplification of the naive binary recursive form of Fibonacci. We have also indicated how this approach can be used in an exposition of various published program transformation rules in this area. Here we exploited the category theoretic notion of a commuting diagram to help explain the similarity of two apparently quite different diagrams. All of this points towards a future foundation for our geometry, both in diagrammatic and algebraic form, in the area of category theory.

6.1 Acknowledgments

I would like to acknowledge my use of Kristoffer Rose and Ross Moore's excellent XY-pic package (3.6,1998/03/06) for the diagrams in this paper. and also my colleagues, Dr. Hugh Gibbons and Dr. Mícheál Mac an Archinnigh for their guidance and direction, particularly in identifying prior work in the area of transforming recursive programs. And finally, a word of thanks to the referees for helpful comments and pointers to related work.

References

[AK82] J. Arsac and Y. Kodratoff. Some techniques for recursion removal from recursive functions. *ACM Trans. on Programming Languages and Systems*, 4(2), April 1982.

[BD77] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67, January 1977.

[Bd97] Richard Bird and Oege de Moor. *Algebra of Programming.* Series in Computer Science. Prentice Hall International, London, 1997.

[BW82] F.L. Bauer and H. Wössner. *Algorithmic Language and Program Development.* Springer Verlag, Berlin, 1982.

[CF58] H. B. Curry and R. Feys. *Combinatory Logic*, volume 1. North Holland, Amsterdam, 1958.

[CL95] Sharon Curtis and Gavin Lowe. A graphical calculus. Technical report, Oxford University Computing Laboratory, 1995. Available from `http://www.comlab.ox.ac.uk`, under `/oucl/publications/books/algebra/papers`, called `graph.ps.gz`.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming.* Automatic Computation. Prentice-Hall, New York, 1976.

[DS89] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics.* Texts and Monographs in Computer Science. Springer Verlag, New York, 1989.

[Eil74] Samuel Eilenberg. *Automata, Languages and Machines*, volume B. Academic Press, New York, 1974.

[HM97] Arthur Hughes and Mícheál Mac an Airchinnigh. The Geometry of Distributions in Formal Methods. In D. Duke and A. S. Evans, editors, *2nd BCS-FACS Northern Formal Methods Workshop*, electronic Workshops in Computing, Ilkley, West Yorkshire, U.K., September 1997. BCS-FACS, Springer Verlag.

[Hug98] Arthur Hughes. Constructive mathematics in topoi. In A. Butterfield and S. Flynn, editors, *2nd Irish Workshop on Formal Methods*, Cork, Ireland, July 1998. Irish Formal Methods Special Interest Group, University College, Cork. Draft Proceedings.

[LS97] F. William Lawvere and Stephen H. Schanuel. *Conceptual Mathematics: a first introduction to categories*. Cambridge University Press, 1997.

[Mac90] Mícheál Mac an Airchinnigh. *Conceptual Models and Computing*. PhD dissertation, University of Dublin, Trinity College, Department of Computer Science, 1990.

[Mac96] Mícheál Mac an Airchinnigh. Towards a New Conceptual Framework for the Modelling of Dynamically Distributed Systems. In D. Duke and A. S. Evans, editors, *BCS-FACS Northern Formal Methods Workshop*, electronic Workshops in Computing, Ilkley, West Yorkshire, U.K., July 1996. BCS-FACS, Springer Verlag.

[Mac98] Mícheál Mac an Airchinnigh, 1998. unpublished comment.

[Mil94] Robin Milner. Pi-nets: a graphical form of π-calculus. In Donald Sannella, editor, *Programming Languages and Systems — ESOP'94*, pages 26–42. Springer Verlag, April 1994.

[Par90] Helmut A. Partsch. *Specification and Transformation of Programs*. Texts and monographs in Computer Science. Springer Verlag, Berlin, 1990.

[Par95] Joachim Parrow. Interaction diagrams. *Nordic Journal of Computing*, 2:407–443, 1995.

[PP86] P. Pepper and H. Partsch. Program transformations expressed by algebraic type manipulations. *Technology and Science of Informatics*, 5(3):pp197–212, 1986.

[Wad87] P. Wadler. *Views: a way for pattern matching to cohabit with data abstraction*, pages 307–313. Association for Computing Machinery, 1987.

[Wal91] R. F. C. Walters. *Categories and Computer Science*. Cambridge Computer Science Texts. Cambridge University Press, 1991.