J. Functional Programming **9** (4): 355–372, July 1999 © 1999 Cambridge University Press 1

A tutorial on the universality and expressiveness of fold

Graham Hutton
University of Nottingham, UK
`http://www.cs.nott.ac.uk/~gmh`

Abstract

In functional programming, *fold* is a standard operator that encapsulates a simple pattern of recursion for processing lists. This article is a tutorial on two key aspects of the fold operator for lists. First of all, we emphasize the use of the *universal property* of fold both as a *proof principle* that avoids the need for inductive proofs, and as a *definition principle* that guides the transformation of recursive functions into definitions using fold. Secondly, we show that even though the pattern of recursion encapsulated by fold is simple, in a language with tuples and functions as first-class values the fold operator has greater *expressive power* than might first be expected.

1 Introduction

Many programs that involve repetition are naturally expressed using some form of recursion, and properties proved of such programs using some form of induction. Indeed, in the functional approach to programming, recursion and induction are the primary tools for defining and proving properties of programs.

Not surprisingly, many recursive programs will share a common pattern of recursion, and many inductive proofs will share a common pattern of induction. Repeating the same patterns again and again is tedious, time consuming, and prone to error. Such repetition can be avoided by introducing special *recursion operators* and *proof principles* that encapsulate the common patterns, allowing us to concentrate on the parts that are different for each application.

In functional programming, *fold* (also known as *foldr*) is a standard recursion operator that encapsulates a common pattern of recursion for processing lists. The fold operator comes equipped with a proof principle called *universality*, which encapsulates a common pattern of inductive proof concerning lists. Fold and its universal property together form the basis of a simple but powerful calculational theory of programs that process lists. This theory generalises from lists to a variety of other datatypes, but for simplicity we restrict our attention to lists.

This article is a tutorial on two key aspects of the fold operator for lists. First of all, we emphasize the use of the universal property of fold (together with the derived *fusion property*) both as *proof principles* that avoid the need for inductive proofs, and as *definition principles* that guide the transformation of recursive functions into definitions using fold. Secondly, we show that even though the pattern of recursion encapsulated by fold is simple, in a language with tuples and functions as first-class values the fold operator has greater *expressive power* than might first be expected, thus permitting the powerful universal and fusion properties of fold to be applied to a larger class of programs. The article concludes with a survey of other work on recursion operators that we do not have space to pursue here.

The article is aimed at a reader who is familiar with the basics of functional programming, say to the level of (Bird & Wadler, 1988; Bird, 1998). All programs in the article are written in Haskell (Peterson *et al.* , 1997), the standard lazy functional programming language. However, no special features of Haskell are used, and the ideas can easily be adapted to other functional languages.

2 The fold operator

The *fold* operator has its origins in recursion theory (Kleene, 1952), while the use of *fold* as a central concept in a programming language dates back to the reduction operator of APL (Iverson, 1962), and later to the insertion operator of FP (Backus, 1978). In Haskell, the *fold* operator for lists can be defined as follows:

$$
\begin{array}{lcl}
\mathit{fold} & :: & (\alpha \to \beta \to \beta) \to \beta \to ([\alpha] \to \beta) \\
\mathit{fold}\ f\ v\ \ [\,] & = & v \\
\mathit{fold}\ f\ v\ (x : xs) & = & f\ x\ (\mathit{fold}\ f\ v\ xs)
\end{array}
$$

That is, given a function f of type $\alpha \to \beta \to \beta$ and a value v of type β, the function $\mathit{fold}\ f\ v$ processes a list of type $[\alpha]$ to give a value of type β by replacing the *nil* constructor $[\,]$ at the end of the list by the value v, and each *cons* constructor $(:)$ within the list by the function f. In this manner, the *fold* operator encapsulates a simple pattern of recursion for processing lists, in which the two constructors for lists are simply replaced by other values and functions. A number of familiar functions on lists have a simple definition using *fold*. For example:

$$
\begin{array}{lcl@{\qquad\qquad}lcl}
\mathit{sum} & :: & [\mathit{Int}] \to \mathit{Int} & \mathit{product} & :: & [\mathit{Int}] \to \mathit{Int} \\
\mathit{sum} & = & \mathit{fold}\ (+)\ 0 & \mathit{product} & = & \mathit{fold}\ (\times)\ 1 \\
\\
\mathit{and} & :: & [\mathit{Bool}] \to \mathit{Bool} & \mathit{or} & :: & [\mathit{Bool}] \to \mathit{Bool} \\
\mathit{and} & = & \mathit{fold}\ (\wedge)\ \mathit{True} & \mathit{or} & = & \mathit{fold}\ (\vee)\ \mathit{False}
\end{array}
$$

Recall that enclosing an infix operator $\oplus$ in parentheses $(\oplus)$ converts the operator into a prefix function. This notational device, called *sectioning*, is often useful when defining simple functions using *fold*. If required, one of the arguments to the operator can also be enclosed in the parentheses. For example, the function $(+\!\!+)$ that appends two lists to give a single list can be defined as follows:

$$
\begin{array}{lcl}
(+\!\!+) & :: & [\alpha] \to [\alpha] \to [\alpha] \\
(+\!\!+\ ys) & = & \mathit{fold}\ (:)\ ys
\end{array}
$$

In all our examples so far, the constructor $(:)$ is replaced by a built-in function. However, in most applications of *fold* the constructor $(:)$ will be replaced by a user-defined function, often defined as a nameless function using the λ notation, as in the following definitions of standard list-processing functions:

$$
\begin{array}{lcl}
\mathit{length} & :: & [\alpha] \rightarrow \mathit{Int} \\
\mathit{length} & = & \mathit{fold}\ \ (\lambda x\ n \rightarrow 1 + n)\ \ 0 \\
\\
\mathit{reverse} & :: & [\alpha] \rightarrow [\alpha] \\
\mathit{reverse} & = & \mathit{fold}\ \ (\lambda x\ \mathit{xs} \rightarrow \mathit{xs} +\!\!+ [x])\ \ [\,] \\
\\
\mathit{map} & :: & (\alpha \rightarrow \beta) \rightarrow ([\alpha] \rightarrow [\beta]) \\
\mathit{map}\ f & = & \mathit{fold}\ \ (\lambda x\ \mathit{xs} \rightarrow f\ x\ :\ \mathit{xs})\ \ [\,] \\
\\
\mathit{filter} & :: & (\alpha \rightarrow \mathit{Bool}) \rightarrow ([\alpha] \rightarrow [\alpha]) \\
\mathit{filter}\ p & = & \mathit{fold}\ \ (\lambda x\ \mathit{xs} \rightarrow \mathbf{if}\ \ p\ \ x\ \ \mathbf{then}\ \ x : \mathit{xs}\ \ \mathbf{else}\ \ \mathit{xs})\ \ [\,]
\end{array}
$$

Programs written using *fold* can be less readable than programs written using explicit recursion, but can be constructed in a systematic manner, and are better suited to transformation and proof. For example, we will see later on in the article how the above definition for *map* using *fold* can be constructed from the standard definition using explicit recursion, and more importantly, how the definition using *fold* simplifies the process of proving properties of the *map* function.

3 The universal property of fold

As with the *fold* operator itself, the universal property of *fold* also has its origins in recursion theory. The first systematic use of the universal property in functional programming was by Malcolm (1990a), in his generalisation of Bird and Meerten's theory of lists (Bird, 1989; Meertens, 1983) to arbitrary regular datatypes. For finite lists, the universal property of *fold* can be stated as the following equivalence between two definitions for a function g that processes lists:

$$
\begin{array}{lcl}
g\ \ [\,] & = & v \\
g\ \ (x : \mathit{xs}) & = & f\ \ x\ \ (g\ \ \mathit{xs})
\end{array}
\quad \Leftrightarrow \quad g \ = \ \mathit{fold}\ \ f\ \ v
$$

In the right-to-left direction, substituting $g = \mathit{fold}\ \ f\ \ v$ into the two equations for g gives the recursive definition for *fold*. Conversely, in the left-to-right direction the two equations for g are precisely the assumptions required to show that $g = \mathit{fold}\ \ f\ \ v$ using a simple proof by induction on finite lists (Bird, 1998). Taken as a whole, the universal property states that for finite lists the function $\mathit{fold}\ \ f\ \ v$ is not just a solution to its defining equations, but in fact the *unique* solution.

The key to the utility of the universal property is that it makes explicit the two assumptions required for a certain pattern of inductive proof. For specific cases then, by verifying the two assumptions (which can typically be done without the need for induction) we can then appeal to the universal property to complete the inductive proof that $g = \mathit{fold}\ \ f\ \ v$. In this manner, the universal property of *fold* encapsulates a simple pattern of inductive proof concerning lists, just as the *fold* operator itself encapsulates a simple pattern of recursion for processing lists.

The universal property of *fold* can be generalised to handle partial and infinite lists (Bird, 1998), but for simplicity we only consider finite lists in this article.

3.1 Universality as a proof principle

The primary application of the universal property of *fold* is as a proof principle that avoids the need for inductive proofs. As a simple first example, consider the following equation between functions that process a list of numbers:

$$(+1) \cdot sum \;=\; fold\ (+)\ 1$$

The left-hand function sums a list and then increments the result. The right-hand function processes a list by replacing each (:) by the addition function (+) and the empty list [] by the constant 1. The equation asserts that these two functions always give the same result when applied to the same list.

To prove the above equation, we begin by observing that it matches the right-hand side $g = fold\ f\ v$ of the universal property of *fold*, with $g = (+1) \cdot sum$, $f = (+)$, and $v = 1$. Hence, by appealing to the universal property, we conclude that the equation to be proved is equivalent to the following two equations:

$$\begin{array}{lcl} ((+1) \cdot sum)\ [\,] & = & 1 \\ ((+1) \cdot sum)\ (x : xs) & = & (+)\ x\ (((+1) \cdot sum)\ xs) \end{array}$$

At first sight, these may seem more complicated than the original equation. However, simplifying using the definitions of composition and sectioning gives

$$\begin{array}{lcl} sum\ [\,] + 1 & = & 1 \\ sum\ (x : xs) + 1 & = & x + (sum\ xs + 1) \end{array}$$

which can now be verified by simple calculations, shown here in two columns:

$$\begin{array}{cl} & sum\ [\,] + 1 \\ = & \quad \{ \text{ Definition of } sum \ \} \\ & 0 + 1 \\ = & \quad \{ \text{ Arithmetic } \} \\ & 1 \end{array} \qquad\qquad \begin{array}{cl} & sum\ (x : xs) + 1 \\ = & \quad \{ \text{ Definition of } sum \ \} \\ & (x + sum\ xs) + 1 \\ = & \quad \{ \text{ Arithmetic } \} \\ & x + (sum\ xs + 1) \end{array}$$

This completes the proof. Normally this proof would have required an explicit use of induction. However, in the above proof the use of induction has been encapsulated in the universal property of *fold*, with the result that the proof is reduced to a simplification step followed by two simple calculations.

In general, any two functions on lists that can be proved equal by induction can also be proved equal using the universal property of the *fold* operator, provided, of course, that the functions can be expressed using *fold*. The expressive power of the *fold* operator will be addressed later on in the article.

3.2 The fusion property of fold

Now let us generalise from the *sum* example and consider the following equation between functions that process a list of values:

$$h \cdot fold\ g\ w \;=\; fold\ f\ v$$

This pattern of equation occurs frequently when reasoning about programs written using *fold*. It is not true in general, but we can use the universal property of *fold* to calculate conditions under which the equation will indeed be true. The equation matches the right-hand side of the universal property, from which we conclude that the equation is equivalent to the following two equations:

$$\begin{array}{lcl} (h \;\cdot\; \mathit{fold}\;\; g\;\; w)\;\;\; [\,] & = & v \\ (h \;\cdot\; \mathit{fold}\;\; g\;\; w)\;\; (x : xs) & = & f\;\; x\;\; ((h \;\cdot\; \mathit{fold}\;\; g\;\; w)\;\; xs) \end{array}$$

Simplifying using the definition of composition gives

$$\begin{array}{lcl} h\;\; (\mathit{fold}\;\; g\;\; w\;\;\;\; [\,]) & = & v \\ h\;\; (\mathit{fold}\;\; g\;\; w\;\; (x : xs)) & = & f\;\; x\;\; (h\;\; (\mathit{fold}\;\; g\;\; w\;\; xs)) \end{array}$$

which can now be further simplified by two calculations:

$$\begin{array}{ll} & h\;\; (\mathit{fold}\;\; g\;\; w\;\; [\,]) \;=\; v \\ \Leftrightarrow & \quad \{ \text{ Definition of } \mathit{fold} \text{ } \} \\ & h\;\; w \;=\; v \end{array}$$

and

$$\begin{array}{ll} & h\;\; (\mathit{fold}\;\; g\;\; w\;\; (x : xs)) \;=\; f\;\; x\;\; (h\;\; (\mathit{fold}\;\; g\;\; w\;\; xs)) \\ \Leftrightarrow & \quad \{ \text{ Definition of } \mathit{fold} \text{ } \} \\ & h\;\; (g\;\; x\;\; (\mathit{fold}\;\; g\;\; w\;\; xs)) \;=\; f\;\; x\;\; (h\;\; (\mathit{fold}\;\; g\;\; w\;\; xs)) \\ \Leftarrow & \quad \{ \text{ Generalising } (\mathit{fold}\;\; g\;\; w\;\; xs) \text{ to a fresh variable } y \text{ } \} \\ & h\;\; (g\;\; x\;\; y) \;=\; f\;\; x\;\; (h\;\; y) \end{array}$$

That is, using the universal property of *fold* we have calculated—without an explicit use of induction—two simple conditions that are together sufficient to ensure for all finite lists that the composition of an arbitrary function and a *fold* can be fused together to give a single *fold*. Following this interpretation, this property is called the *fusion* property of the *fold* operator, and can be stated as follows:

$$\begin{array}{lcl} h\;\; w & = & v \\ h\;\; (g\;\; x\;\; y) & = & f\;\; x\;\; (h\;\; y) \end{array} \quad\Rightarrow\quad h \;\cdot\; \mathit{fold}\;\; g\;\; w \;\;=\;\; \mathit{fold}\;\; f\;\; v$$

The first systematic use of the fusion property in functional programming was again by Malcolm (1990a), generalising earlier work by Bird and Meertens (Bird, 1989; Meertens, 1983). As with the universal property, the primary application of the fusion property is as a proof principle that avoids the need for inductive proofs. In fact, for many practical examples the fusion property is often preferable to the universal property. As a simple first example, consider again the equation:

$$(+1) \;\cdot\; \mathit{sum} \;\;=\;\; \mathit{fold}\;\; (+)\;\; 1$$

In the previous section this equation was proved using the universal property of *fold*. However, the proof is simpler using the fusion property. First of all, we replace the function *sum* by its definition using *fold* given earlier:

$$(+1) \;\cdot\; \mathit{fold}\;\; (+)\;\; 0 \;\;=\;\; \mathit{fold}\;\; (+)\;\; 1$$

The equation now matches the conclusion of the fusion property, from which we conclude that the equation follows from the following two assumptions:

$$\begin{array}{lcl} (+1)\ 0 & = & 1 \\ (+1)\ ((+)\ x\ y) & = & (+)\ x\ ((+1)\ y) \end{array}$$

Simplifying these equations using the definition of sectioning gives $0 + 1 = 1$ and $(x + y) + 1 = x + (y + 1)$, which are true by simple properties of arithmetic. More generally, by replacing the use of addition in this example by an arbitrary infix operator $\oplus$ that is associative, a simple application of fusion shows that:

$$(\oplus\ a)\ \cdot\ \mathit{fold}\ (\oplus)\ b \quad = \quad \mathit{fold}\ (\oplus)\ (b \oplus a)$$

For a more interesting example, consider the following well-known equation, which asserts that the *map* operator distributes over function composition $(\cdot)$:

$$\mathit{map}\ f\ \cdot\ \mathit{map}\ g \quad = \quad \mathit{map}\ (f \cdot g)$$

By replacing the second and third occurrences of the *map* operator in the equation by its definition using *fold* given earlier, the equation can be rewritten in a form that matches the conclusion of the fusion property:

$$\begin{array}{c} \mathit{map}\ f\ \cdot\ \mathit{fold}\ (\lambda x\ xs \rightarrow g\ x\ :\ xs)\ [\,] \\ = \\ \mathit{fold}\ (\lambda x\ xs \rightarrow (f \cdot g)\ x\ :\ xs)\ [\,] \end{array}$$

Appealing to the fusion property and then simplifying gives the following two equations, which are trivially true by the definitions of *map* and $(\cdot)$:

$$\begin{array}{lcl} \mathit{map}\ f\ \ [\,] & = & [\,] \\ \mathit{map}\ f\ (g\ x\ :\ y) & = & (f \cdot g)\ x\ :\ \mathit{map}\ f\ y \end{array}$$

In addition to the fusion property, there are a number of other useful properties of the *fold* operator that can be derived from the universal property (Bird, 1998). However, the fusion property suffices for many practical cases, and one can always revert to the full power of the universal property if fusion is not appropriate.

3.3 Universality as a definition principle

As well as being used as a proof principle, the universal property of *fold* can also be used as a definition principle that guides the transformation of recursive functions into definitions using *fold*. As a simple first example, consider the recursively defined function *sum* that calculates the sum of a list of numbers:

$$\begin{array}{lcl} \mathit{sum} & :: & [\mathit{Int}] \rightarrow \mathit{Int} \\ \mathit{sum}\ \ [\,] & = & 0 \\ \mathit{sum}\ (x : xs) & = & x\ +\ \mathit{sum}\ xs \end{array}$$

Suppose now that we want to redefine *sum* using *fold*. That is, we want to solve the equation $\mathit{sum} = \mathit{fold}\ \ f\ \ v$ for a function f and a value v. We begin by observing that the equation matches the right-hand side of the universal property, from which we conclude that the equation is equivalent to the following two equations:

$$\begin{array}{lcl} \mathit{sum}\ \ [\,] & = & v \\ \mathit{sum}\ (x : xs) & = & f\ \ x\ \ (\mathit{sum}\ \ xs) \end{array}$$

From the first equation and the definition of *sum*, it is immediate that $v = 0$. From the second equation, we calculate a definition for f as follows:

$$\begin{array}{ll} & sum\ (x : xs) \ = \ f\ x\ (sum\ xs) \\ \Leftrightarrow & \quad \{ \text{ Definition of } sum \ \} \\ & x \ + \ sum\ xs \ = \ f\ x\ (sum\ xs) \\ \Leftarrow & \quad \{ \dagger \text{ Generalising } (sum\ xs) \text{ to } y \ \} \\ & x \ + \ y \ = \ f\ x\ y \\ \Leftrightarrow & \quad \{ \text{ Functions } \} \\ & f \ = \ (+) \end{array}$$

That is, using the universal property we have calculated that:

$$sum \quad = \quad fold\ (+)\ 0$$

Note that the key step (†) above in calculating a definition for f is the generalisation of the expression $sum\ xs$ to a fresh variable y. In fact, such a generalisation step is not specific to the *sum* function, but will be a key step in the transformation of any recursive function into a definition using *fold* in this manner.

Of course, the *sum* example above is rather artificial, because the definition of *sum* using *fold* is immediate. However, there are many examples of functions whose definition using *fold* is not so immediate. For example, consider the recursively defined function $map\ f$ that applies a function f to each element of a list:

$$\begin{array}{lcl} map & :: & (\alpha \to \beta) \to ([\alpha] \to [\beta]) \\ map\ f\ \ [\,] & = & [\,] \\ map\ f\ (x : xs) & = & f\ x : map\ f\ xs \end{array}$$

To redefine $map\ f$ using *fold* we must solve the equation $map\ f = fold\ \ g\ \ v$ for a function g and a value v. By appealing to the universal property, we conclude that this equation is equivalent to the following two equations:

$$\begin{array}{lcl} map\ f\ \ [\,] & = & v \\ map\ f\ (x : xs) & = & g\ x\ (map\ f\ xs) \end{array}$$

From the first equation and the definition of *map* it is immediate that $v = [\,]$. From the second equation, we calculate a definition for g as follows:

$$\begin{array}{ll} & map\ f\ (x : xs) \ = \ g\ x\ (map\ f\ xs) \\ \Leftrightarrow & \quad \{ \text{ Definition of } map \ \} \\ & f\ x : map\ f\ xs \ = \ g\ x\ (map\ f\ xs) \\ \Leftarrow & \quad \{ \text{ Generalising } (map\ f\ xs) \text{ to } ys \ \} \\ & f\ x : ys \ = \ g\ x\ ys \\ \Leftrightarrow & \quad \{ \text{ Functions } \} \\ & g \ = \ \lambda x\ ys \to f\ x : ys \end{array}$$

That is, using the universal property we have calculated that:

$$map\ f \quad = \quad fold\ (\lambda x\ ys \to f\ x : ys)\ [\,]$$

In general, any function on lists that can be expressed using the *fold* operator can be transformed into such a definition using the universal property of *fold*.

4 Increasing the power of fold: generating tuples

As a simple first example of the use of *fold* to generate tuples, consider the function *sumlength* that calculates the sum and length of a list of numbers:

$$\begin{array}{lcl} sumlength & :: & [Int] \to (Int, Int) \\ sumlength\ xs & = & (sum\ xs,\ length\ xs) \end{array}$$

By a straightforward combination of the definitions of the functions *sum* and *length* using *fold* given earlier, the function *sumlength* can be redefined as a single application of *fold* that generates a pair of numbers from a list of numbers:

$$sumlength \quad = \quad fold\ \ (\lambda n\ (x, y) \to (n + x,\ 1 + y))\ \ (0, 0)$$

This definition is more efficient than the original definition, because it only makes a single traversal over the argument list, rather than two separate traversals. Generalising from this example, any pair of applications of *fold* to the same list can always be combined to give a single application of *fold* that generates a pair, by appealing to the so-called 'banana split' property of *fold* (Meijer, 1992). The strange name of this property derives from the fact that the *fold* operator is sometimes written using brackets $(\!|\ \ |\!)$ that resemble bananas, and the pairing operator is sometimes called split. Hence, their combination can be termed a banana split!

As a more interesting example, let us consider the function *dropWhile* p that removes initial elements from a list while all the elements satisfy the predicate p:

$$\begin{array}{lcl} dropWhile & :: & (\alpha \to Bool) \to ([\alpha] \to [\alpha]) \\ dropWhile\ p\ \ [\,] & = & [\,] \\ dropWhile\ p\ (x : xs) & = & \mathbf{if}\ p\ x\ \mathbf{then}\ dropWhile\ p\ xs\ \mathbf{else}\ x : xs \end{array}$$

Suppose now that we want to redefine *dropWhile* p using the *fold* operator. By appealing to the universal property, we conclude that the equation *dropWhile* $p =$ *fold* $f\ \ v$ is equivalent to the following two equations:

$$\begin{array}{lcl} dropWhile\ p\ \ [\,] & = & v \\ dropWhile\ p\ (x : xs) & = & f\ x\ (dropWhile\ p\ xs) \end{array}$$

From the first equation it is immediate that $v = [\,]$. From the second equation, we attempt to calculate a definition for f in the normal manner:

$$\begin{array}{ll} & dropWhile\ p\ (x : xs) = f\ x\ (dropWhile\ p\ xs) \\ \Leftrightarrow & \quad \{\ \text{Definition of } dropWhile\ \} \\ & \mathbf{if}\ p\ x\ \mathbf{then}\ dropWhile\ p\ xs\ \mathbf{else}\ x : xs = f\ x\ (dropWhile\ p\ xs) \\ \Leftarrow & \quad \{\ \text{Generalising } (dropWhile\ p\ xs) \text{ to } ys\ \} \\ & \mathbf{if}\ p\ x\ \mathbf{then}\ ys\ \mathbf{else}\ x : xs = f\ x\ ys \end{array}$$

Unfortunately, the final line above is not a valid definition for f, because the variable xs occurs freely. In fact, it is not possible to redefine *dropWhile* p directly using *fold*. However, it is possible indirectly, because the more general function

$$\begin{array}{lcl} dropWhile' & :: & (\alpha \to Bool) \to ([\alpha] \to ([\alpha],\ [\alpha])) \\ dropWhile'\ p\ xs & = & (dropWhile\ p\ xs,\ xs) \end{array}$$

that pairs up the result of applying $dropWhile\ p$ to a list with the list itself *can* be redefined using *fold*. By appealing to the universal property, we conclude that the equation $dropWhile'\ p = fold\ f\ v$ is equivalent to the following two equations:

$$\begin{array}{lll} dropWhile'\ p\ [\,] & = & v \\ dropWhile'\ p\ (x : xs) & = & f\ x\ (dropWhile'\ p\ xs) \end{array}$$

A simple calculation from the first equation gives $v = ([\,], [\,])$. From the second equation, we calculate a definition for f as follows:

$$\begin{array}{ll} & dropWhile'\ p\ (x : xs) = f\ x\ (dropWhile'\ p\ xs) \\ \Leftrightarrow & \quad \{\ \text{Definition of } dropWhile'\ \} \\ & (dropWhile\ p\ (x : xs),\ x : xs) = f\ x\ (dropWhile\ p\ xs,\ xs) \\ \Leftrightarrow & \quad \{\ \text{Definition of } dropWhile\ \} \\ & (\mathbf{if}\ p\ x\ \mathbf{then}\ dropWhile\ p\ xs\ \mathbf{else}\ x : xs,\ x : xs) \\ & \quad = f\ x\ (dropWhile\ p\ xs,\ xs) \\ \Leftarrow & \quad \{\ \text{Generalising } (dropWhile\ p\ xs) \text{ to } ys\ \} \\ & (\mathbf{if}\ p\ x\ \mathbf{then}\ ys\ \mathbf{else}\ x : xs,\ x : xs) = f\ x\ (ys, xs) \end{array}$$

Note that the final line above *is* a valid definition for f, because all the variables are bound. In summary, using the universal property we have calculated that:

$$\begin{array}{l} dropWhile'\ p \ = \ fold\ f\ v \\ \quad \mathbf{where} \\ \qquad \begin{array}{lll} f\ x\ (ys, xs) & = & (\mathbf{if}\ p\ x\ \mathbf{then}\ ys\ \mathbf{else}\ x : xs,\ x : xs) \\ v & = & ([\,], [\,]) \end{array} \end{array}$$

This definition satisfies the equation $dropWhile'\ p\ xs = (dropWhile\ p\ xs,\ xs)$, but does not make use of *dropWhile* in its definition. Hence, the function *dropWhile* itself can now be redefined simply by $dropWhile\ p = fst \cdot dropWhile'\ p$.

In conclusion, by first generalising to a function $dropWhile'$ that pairs the desired result with the argument list, we have now shown how the function *dropWhile* can be redefined in terms of *fold*, as required. In fact, this result is an instance of a general theorem (Meertens, 1992) that states that any function on finite lists that is defined by pairing the desired result with the argument list can always be redefined in terms of *fold*, although not always in a way that does not make use of the original (possibly recursive) definition for the function.

4.1 Primitive recursion

In this section we show that by using the tupling technique from the previous section, every primitive recursive function on lists can be redefined in terms of *fold*. Let us begin by recalling that the *fold* operator captures the following simple pattern of recursion for defining a function h that processes lists:

$$\begin{array}{lll} h\ [\,] & = & v \\ h\ (x : xs) & = & g\ x\ (h\ xs) \end{array}$$

Such functions can be redefined by $h = fold\ g\ v$. We will generalise this pattern of recursion to primitive recursion in two steps. First of all, we introduce an extra argument y to the function h, which in the base case is processed by a new function f, and in the recursive case is passed unchanged to the functions g and h. That is, we now consider the following pattern of recursion for defining a function h:

$$\begin{array}{lll} h\ y\ \ [\,] & = & f\ y \\ h\ y\ (x : xs) & = & g\ y\ x\ (h\ y\ xs) \end{array}$$

By simple observation, or a routine application of the universal property of *fold*, the function $h\ y$ can be redefined using *fold* as follows:

$$h\ y \quad = \quad \mathit{fold}\ (g\ y)\ (f\ y)$$

For the second step, we introduce the list xs as an extra argument to the auxiliary function g. That is, we now consider the following pattern for defining h:

$$\begin{array}{lll} h\ y\ \ [\,] & = & f\ y \\ h\ y\ (x : xs) & = & g\ y\ x\ xs\ (h\ y\ xs) \end{array}$$

This pattern of recursion on lists is called *primitive recursion* (Kleene, 1952). Technically, the standard definition of primitive recursion requires that the argument y is a finite sequence of arguments. However, because tuples are first-class values in Haskell, treating the case of a single argument y is sufficient.

In order to redefine primitive recursive functions in terms of *fold*, we must solve the equation $h\ y = \mathit{fold}\ i\ j$ for a function i and a value j. This is not possible directly, but is possible indirectly, because the more general function

$$k\ y\ xs \quad = \quad (h\ y\ xs,\ xs)$$

that pairs up the result of applying $h\ y$ to a list with the list itself *can* be redefined using *fold*. By appealing to the universal property of *fold*, we conclude that the equation $k\ y = \mathit{fold}\ i\ j$ is equivalent to the following two equations:

$$\begin{array}{lll} k\ y\ \ [\,] & = & j \\ k\ y\ (x : xs) & = & i\ x\ (k\ y\ xs) \end{array}$$

A simple calculation from the first equation gives $j = (f\ y,\ [\,])$. From the second equation, we calculate a definition for i as follows:

$$\begin{array}{ll}
 & k\ y\ (x : xs) = i\ x\ (k\ y\ xs) \\
\Leftrightarrow & \quad \{\ \text{Definition of } k\ \} \\
 & (h\ y\ (x : xs),\ x : xs) = i\ x\ (h\ y\ xs,\ xs) \\
\Leftrightarrow & \quad \{\ \text{Definition of } h\ \} \\
 & (g\ y\ x\ xs\ (h\ y\ xs),\ x : xs) = i\ x\ (h\ y\ xs,\ xs) \\
\Leftarrow & \quad \{\ \text{Generalising } (h\ y\ xs) \text{ to } z\ \} \\
 & (g\ y\ x\ xs\ z,\ x : xs) = i\ x\ (z,\ xs)
\end{array}$$

In summary, using the universal property we have calculated that:

$$\begin{array}{lll}
k\ y & = & \mathit{fold}\ i\ j \\
 & & \quad \textbf{where} \\
 & & \qquad i\ x\ (z,\ xs) = (g\ y\ x\ xs\ z,\ x : xs) \\
 & & \qquad j \qquad\qquad\ \ = (f\ y,\ [\,])
\end{array}$$

This definition satisfies the equation $k\ \ y\ \ xs = (h\ \ y\ \ xs,\ xs)$, but does not make use of h in its definition. Hence, the primitive recursive function h itself can now be redefined simply by $h\ \ y = \mathit{fst}\ \cdot\ k\ \ y$. In conclusion, we have now shown how an arbitrary primitive recursive function on lists can be redefined in terms of *fold*.

Note that the use of tupling to define primitive recursive functions in terms of *fold* is precisely the key to defining the predecessor function for the Church numerals (Barendregt, 1984). Indeed, the intuition behind the representation of the natural numbers (or more generally, any inductive datatype) in the λ-calculus is the idea of representing each number by its *fold* operator. For example, the number $3 = \mathit{succ}\ \ (\mathit{succ}\ \ (\mathit{succ}\ \ \mathit{zero}))$ is represented by the term $\lambda f\ \ x \rightarrow f\ (f\ (f\ x))$, which is the *fold* operator for 3 in the sense that the arguments f and x can be viewed as the replacements for the *succ* and *zero* constructors respectively.

5 Using fold to generate functions

Having functions as first-class values increases the power of primitive recursion, and hence the power of the *fold* operator. As a simple first example of the use of *fold* to generate functions, the function *compose* that forms the composition of a list of functions can be defined using *fold* by replacing each (:) in the list by the composition function $(\cdot)$, and the empty list $[\,]$ by the identity function *id*:

$$\begin{array}{lcl} \mathit{compose} & :: & [\alpha \rightarrow \alpha] \rightarrow (\alpha \rightarrow \alpha) \\ \mathit{compose} & = & \mathit{fold}\ \ (\cdot)\ \ \mathit{id} \end{array}$$

As a more interesting example, let us consider the problem of summing a list of numbers. The natural definition for such a function, $\mathit{sum} = \mathit{fold}\ \ (+)\ \ 0$, processes the numbers in the list in right-to-left order. However, it is also possible to define a function *suml* that processes the numbers in left-to-right order. The *suml* function is naturally defined using an auxiliary function suml' that is itself defined by explicit recursion and makes use of an accumulating parameter n:

$$\begin{array}{lcl} \mathit{suml} & :: & [\mathit{Int}] \rightarrow \mathit{Int} \\ \mathit{suml}\ \ xs & = & \mathit{suml}'\ \ xs\ \ 0 \\ & & \quad \textbf{where} \\ & & \qquad \begin{array}{lllcl} \mathit{suml}' & [\,] & n & = & n \\ \mathit{suml}' & (x : xs) & n & = & \mathit{suml}'\ \ xs\ \ (n + x) \end{array} \end{array}$$

Because the addition function (+) is associative and the constant 0 is unit for addition, the functions *suml* and *sum* always give the same result when applied to the same list. However, the function *suml* has the potential to be more efficient, because it can easily be modified to run in constant space (Bird, 1998).

Suppose now that we want to redefine *suml* using the *fold* operator. This is not possible directly, but is possible indirectly, because the auxiliary function

$$\mathit{suml}' \quad :: \quad [\mathit{Int}] \rightarrow (\mathit{Int} \rightarrow \mathit{Int})$$

can be redefined using *fold*. By appealing to the universal property, we conclude that the equation $\mathit{suml}' = \mathit{fold}\ \ f\ \ v$ is equivalent to the following two equations:

$$\begin{array}{lcl} suml' \;\; [\,] & = & v \\ suml' \;\; (x : xs) & = & f \;\; x \;\; (suml' \;\; xs) \end{array}$$

A simple calculation from the first equation gives $v = id$. From the second equation, we calculate a definition for the function f as follows:

$$\begin{array}{ll} & suml' \;\; (x : xs) \;\; = \;\; f \;\; x \;\; (suml' \;\; xs) \\ \Leftrightarrow & \quad \{ \text{ Functions } \} \\ & suml' \;\; (x : xs) \;\; n \;\; = \;\; f \;\; x \;\; (suml' \;\; xs) \;\; n \\ \Leftrightarrow & \quad \{ \text{ Definition of } suml' \; \} \\ & suml' \;\; xs \;\; (n + x) \;\; = \;\; f \;\; x \;\; (suml' \;\; xs) \;\; n \\ \Leftarrow & \quad \{ \text{ Generalising } (suml' \;\; xs) \text{ to } g \; \} \\ & g \;\; (n + x) \;\; = \;\; f \;\; x \;\; g \;\; n \\ \Leftrightarrow & \quad \{ \text{ Functions } \} \\ & f \;\; = \;\; \lambda x \; g \rightarrow (\lambda n \rightarrow g \;\; (n + x)) \end{array}$$

In summary, using the universal property we have calculated that:

$$suml' \quad = \quad fold \;\; (\lambda x \; g \rightarrow (\lambda n \rightarrow g \;\; (n + x))) \;\; id$$

This definition states that $suml'$ processes a list by replacing the empty list $[\,]$ by the identity function id on lists, and each constructor $(:)$ by the function that takes a number x and a function g, and returns the function that takes an accumulator value n and returns the result of applying g to the new accumulator value $n + x$.

Note that the structuring of the arguments to $suml' \;::\; [Int] \rightarrow (Int \rightarrow Int)$ is crucial to its definition using *fold*. In particular, if the order of the two arguments is swapped or they are supplied as a pair, then the type of $suml'$ means that it can no longer be defined directly using *fold*. In general, some care regarding the structuring of arguments is required when aiming to redefine functions using *fold*. Moreover, at first sight one might imagine that *fold* can only be used to define functions that process the elements of lists in right-to-left order. However, as the definition of $suml'$ using *fold* shows, the order in which the elements are processed depends on the arguments of *fold*, not on *fold* itself.

In conclusion, by first redefining the auxiliary function $suml'$ using *fold*, we have now shown how the function *suml* can be redefined in terms of *fold*, as required:

$$suml \;\; xs \quad = \quad fold \;\; (\lambda x \; g \rightarrow (\lambda n \rightarrow g \;\; (n + x))) \;\; id \;\; xs \;\; 0$$

We end this section by remarking that the use of *fold* to generate functions provides an elegant technique for the implementation of 'attribute grammars' in functional languages (Fokkinga *et al.*, 1991; Swierstra *et al.*, 1998).

5.1 The foldl operator

Now let us generalise from the *suml* example and consider the standard operator *foldl* that processes the elements of a list in left-to-right order by using a function f to combine values, and a value v as the starting value:

$$\begin{array}{lcl} foldl & :: & (\beta \rightarrow \alpha \rightarrow \beta) \rightarrow \beta \rightarrow ([\alpha] \rightarrow \beta) \\ foldl \;\; f \;\; v \;\; [\,] & = & v \\ foldl \;\; f \;\; v \;\; (x : xs) & = & foldl \;\; f \;\; (f \;\; v \;\; x) \;\; xs \end{array}$$

Using this operator, *suml* can be redefined simply by $suml = foldl\ (+)\ 0$. Many other functions can be defined in a simple way using *foldl*. For example, the standard function *reverse* can redefined using *foldl* as follows:

$$\begin{array}{lcl} reverse & :: & [\alpha] \rightarrow [\alpha] \\ reverse & = & foldl\ (\lambda xs\ x \rightarrow x : xs)\ [\,] \end{array}$$

This definition is more efficient than our original definition using *fold*, because it avoids the use of the inefficient append operator $(+\!\!+)$ for lists.

A simple generalisation of the calculation in the previous section for the function *suml* shows how to redefine the function *foldl* in terms of *fold*:

$$foldl\ f\ v\ xs \quad = \quad fold\ (\lambda x\ g \rightarrow (\lambda a \rightarrow g\ (f\ a\ x)))\ id\ xs\ v$$

In contrast, it is not possible to redefine *fold* in terms of *foldl*, due to the fact that *foldl* is strict in the tail of its list argument but *fold* is not. There are a number of useful 'duality theorems' concerning *fold* and *foldl*, and also some guidelines for deciding which operator is best suited to particular applications (Bird, 1998).

5.2 Ackermann's function

For our final example of the power of *fold*, consider the function *ack* that processes two lists of integers, and is defined using explicit recursion as follows:

$$\begin{array}{llll} ack & & :: & [Int] \rightarrow ([Int] \rightarrow [Int]) \\ ack\ \ [\,] & ys & = & 1 : ys \\ ack\ (x : xs) & [\,] & = & ack\ xs\ [1] \\ ack\ (x : xs) & (y : ys) & = & ack\ xs\ (ack\ (x : xs)\ ys) \end{array}$$

This is Ackermann's function, converted to operate on lists rather than natural numbers by representing each number n by a list with n arbitrary elements. This function is the classic example of a function that is not primitive recursive in a first-order programming language. However, in a higher-order language such as Haskell, Ackermann's function is indeed primitive recursive (Reynolds, 1985). In this section we show how to calculate the definition *ack* in terms of *fold*.

First of all, by appealing to the universal property of *fold*, the equation $ack = fold\ f\ v$ is equivalent to the following two equations:

$$\begin{array}{lcl} ack\ \ [\,] & = & v \\ ack\ (x : xs) & = & f\ x\ (ack\ xs) \end{array}$$

A simple calculation from the first equation gives the definition $v = (1\,:)$. From the second equation, proceeding in the normal manner does not result in a definition for the function f, as the reader may wish to verify. However, progress can be made by first using *fold* to redefine the function $ack\ (x : xs)$ on the left-hand side of the second equation. By appealing to the universal property, the equation $ack\ (x : xs) = fold\ g\ w$ is equivalent to the following two equations:

$$\begin{array}{lcl} ack\ (x : xs)\ \ [\,] & = & w \\ ack\ (x : xs)\ (y : ys) & = & g\ y\ (ack\ (x : xs)\ ys) \end{array}$$

The first equation gives $w = ack\ xs\ [1]$, and from the second:

$$
\begin{array}{cl}
 & ack\ (x:xs)\ (y:ys)\ =\ g\ y\ (ack\ (x:xs)\ ys) \\
\Leftrightarrow & \quad \{\ \text{Definition of } ack\ \} \\
 & ack\ xs\ (ack\ (x:xs)\ ys)\ =\ g\ y\ (ack\ (x:xs)\ ys) \\
\Leftarrow & \quad \{\ \text{Generalising } (ack\ (x:xs)\ ys) \text{ to } zs\ \} \\
 & ack\ xs\ zs\ =\ g\ y\ zs \\
\Leftrightarrow & \quad \{\ \text{Functions}\ \} \\
 & g\ =\ \lambda y \rightarrow ack\ xs
\end{array}
$$

That is, using the universal property we have calculated that:

$$ack\ (x:xs) \quad = \quad fold\ (\lambda y \rightarrow ack\ xs)\ (ack\ xs\ [1])$$

Using this result, we can now calculate a definition for f:

$$
\begin{array}{cl}
 & ack\ (x:xs)\ =\ f\ x\ (ack\ xs) \\
\Leftrightarrow & \quad \{\ \text{Result above}\ \} \\
 & fold\ (\lambda y \rightarrow ack\ xs)\ (ack\ xs\ [1])\ =\ f\ x\ (ack\ xs) \\
\Leftarrow & \quad \{\ \text{Generalising } (ack\ xs) \text{ to } g\ \} \\
 & fold\ (\lambda y \rightarrow g)\ (g\ [1])\ =\ f\ x\ g \\
\Leftrightarrow & \quad \{\ \text{Functions}\ \} \\
 & f\ =\ \lambda x\ g \rightarrow fold\ (\lambda y \rightarrow g)\ (g\ [1])
\end{array}
$$

In summary, using the universal property *twice* we have calculated that:

$$ack \quad = \quad fold\ (\lambda x\ g \rightarrow fold\ (\lambda y \rightarrow g)\ (g\ [1]))\ (1\ :)$$

6 Other work on recursion operators

In this final section we briefly survey a selection of other work on recursion operators that we did not have space to pursue in this article.

Fold for regular datatypes. The fold operator is not specific to lists, but can be generalised in a uniform way to 'regular' datatypes. Indeed, using ideas from category theory, a single fold operator can be defined that can be used with any regular datatype (Malcolm, 1990b; Meijer *et al.*, 1991; Sheard & Fegaras, 1993).

Fold for nested datatypes. The fold operator can also be generalised in a natural way to 'nested' datatypes. However, the resulting operator appears to be too general to be widely useful. Finding solutions to this problem is the subject of current research (Bird & Meertens, 1998; Jones & Blampied, 1998).

Fold for functional datatypes. Generalising the fold operator to datatypes that involve functions gives rise to technical problems, due to the contravariant nature of function types. Using ideas from category theory, a fold operator can be defined that works for such datatypes (Meijer & Hutton, 1995), but the use of this operator is not well understood, and practical applications are lacking. However, a simpler but less general solution has given rise to some interesting applications concerning cyclic structures (Fegaras & Sheard, 1996).

Monadic fold. In a series of influential articles, Wadler showed how pure functional programs that require imperative features such as state and exceptions can be modelled using monads (Wadler, 1990; Wadler, 1992a; Wadler, 1992b). Building on this work, the notion of a 'monadic fold' combines the use of fold operators to structure the processing of recursive values with the use of monads to structure the use of imperative features (Fokkinga, 1994; Meijer & Jeuring, 1995).

Relational fold. The fold operator can also be generalised in a natural way from functions to relations. This generalisation supports the use of fold as a specification construct, in addition to its use as a programming construct. For example, a relational fold is used in the circuit design calculus Ruby (Jones & Sheeran, 1990; Jones, 1990), the Eindhoven spec calculus (Aarts *et al.*, 1992), and in a recent textbook on the algebra of programming (Bird & de Moor, 1997).

Other recursion operators. The fold operator is not the only useful recursion operator. For example, the dual operator unfold for constructing rather than processing recursive values has been used for specification purposes (Jones, 1990; Bird & de Moor, 1997), to program reactive systems (Kieburtz, 1998), to program operational semantics (Hutton, 1998), and is the subject of current research. Other interesting recursion operators include the so-called paramorphisms (Meertens, 1992), hylomorphisms (Meijer, 1992), and zygomorphisms (Malcolm, 1990a).

Automatic program transformation. Writing programs using recursion operators can simplify the process of optimisation during compilation. For example, eliminating the use of intermediate data structures in programs (deforestation) in considerably simplified when programs are written using recursion operators rather than general recursion (Wadler, 1981; Launchbury & Sheard, 1995; Takano & Meijer, 1995). A generic system for transforming programs written using recursion operators is currently under development (de Moor & Sittampalan, 1998).

Polytypic programming. Defining programs that are not specific to particular datatypes has given rise to a new field, called polytypic programming (Backhouse *et al.*, 1998). Formally, a polytypic program is one that is parameterised by one or more datatypes. Polytypic programs have already been defined for a number of applications, including pattern matching (Jeuring, 1995), unification (Jansson & Jeuring, 1998), and various optimisation problems (Bird & de Moor, 1997).

Programming languages. A number of experimental programming languages have been developed that focus on the use of recursion operators rather than general recursion. Examples include the algebraic design language ADL (Kieburtz & Lewis, 1994), the categorical programming language Charity (Cockett & Fukushima, 1992), and the polytypic programming language PolyP (Jansson & Jeuring, 1997).

7 Acknowledgements

I would like to thank Erik Meijer and the members of the Languages and Programming group in Nottingham for many hours of interesting discussions about *fold*. I am also grateful to Roland Backhouse, Mark P. Jones, Philip Wadler, and the anonymous JFP referees for their detailed comments on the article, which led to a substantial improvement in both the content and presentation. This work is supported by Engineering and Physical Sciences Research Council (EPSRC) research grant GR/L74491, Structured Recursive Programming.

References

Aarts, Chritiene, Backhouse, Roland, Hoogendijk, Paul, Voermans, Ed, & van der Woude, Jaap. (1992). *A relational theory of datatypes.* Available on the World-Wide-Web from `http://www.win.tue.nl/win/cs/wp/papers/papers.html`.

Backhouse, Roland, Jansson, Patrik, Jeuring, Johan, & Meertens, Lambert. 1998 (Sept.). Generic programming: An introduction. *Lecture Notes of the 3rd International Summer School on Advanced Functional Programming.*

Backus, John. (1978). Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs. *Cacm*, **9**(Aug.).

Barendregt, Henk. (1984). *The lambda calculus – it's syntax and semantics.* North-Holland. Revised edition.

Bird, Richard. (1989). Constructive functional programming. *Proc. Marktoberdorf International Summer School on Constructive Methods in Computer Science.* Springer-Verlag.

Bird, Richard. (1998). *Introduction to functional programming using Haskell (second edition).* Prentice Hall.

Bird, Richard, & de Moor, Oege. (1997). *Algebra of programming.* Prentice Hall.

Bird, Richard, & Meertens, Lambert. (1998). Nested datatypes. Jeuring, Johan (ed), *Proc. Fourth International Conference on Mathematics of Program Construction.* LNCS, vol. 1422. Springer-Verlag.

Bird, Richard, & Wadler, Philip. (1988). *An introduction to functional programming.* Prentice Hall.

Cockett, Robin, & Fukushima, Tom. (1992). *About Charity.* Yellow Series Report No. 92/480/18. Department of Computer Science, The University of Calgary.

de Moor, Oege, & Sittampalan, Ganesh. 1998 (Sept.). Generic program transformation. *Lecture Notes of the 3rd International Summer School on Advanced Functional Programming.*

Fegaras, Leonidas, & Sheard, Tim. (1996). Revisiting catemorphisms over datatypes with embedded functions. *Proc. 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.*

Fokkinga, Maarten. 1994 (June). *Monadic maps and folds for arbitrary datatypes.* Memoranda Informatica 94-28. University of Twente.

Fokkinga, Maarten, Jeuring, Johan, Meertens, Lambert, & Meijer, Erik. (1991). Translating attribute grammars into catamorphisms. *The Squiggolist*, **2**(1).

Hutton, Graham. (1998). Fold and unfold for program semantics. *Proc. 3rd ACM SIGPLAN International Conference on Functional Programming.*

Iverson, Kenneth E.(1962). *A Programming Language.* Wiley, New York.

Jansson, Patrick, & Jeuring, Johan. (1998). *Polytypic unification.* To appear in the Journal of Functional Programming.

Jansson, Patrik, & Jeuring, Johan. (1997). PolyP - a polytypic programming language extension. *Proc. 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM Press.

Jeuring, Johan. (1995). Polytypic pattern matching. *Proc. 7th International Conference on Functional Programming and Computer Architecture.* ACM Press, San Diego, California.

Jones, Geraint. (1990). *Designing circuits by calculation.* Technical Report PRG-TR-10-90. Oxford University.

Jones, Geraint, & Sheeran, Mary. (1990). Circuit design in Ruby. Staunstrup (ed), *Formal methods for VLSI design.* Elsevier Science Publications, Amsterdam.

Jones, Mark P., & Blampied, Paul. (1998). *A pragmatic approach to maps and folds for parameterized datatypes.* Submitted for publication.

Kieburtz, Richard B.(1998). Reactive functional programming. *Proc. PROCOMET.* Chapman and Hall.

Kieburtz, Richard B. & Lewis, Jeffrey. (1994). *Algebraic Design Language (preliminary definition)*. Oregon Graduate Institute of Science and Technology.

Kleene, S.C. (1952). *Introduction to metamathematics*. Van Nostrand.

Launchbury, John, & Sheard, Tim. (1995). Warm fusion: Deriving build-catas from recursive definitions. *Proc. 7th International Conference on Functional Programming and Computer Architecture*. ACM Press, San Diego, California.

Malcolm, Grant. (1990a). *Algebraic data types and program transformation*. Ph.D. thesis, Groningen University.

Malcolm, Grant. (1990b). Algebraic data types and program transformation. *Science of Computer Programming*, **14**(2-3), 255–280.

Meertens, Lambert. 1983 (Nov.). Algorithmics: Towards programming as a mathematical activity. *Proc. cwi symposium*.

Meertens, Lambert. (1992). Paramorphisms. *Formal Aspects of Computing*, **4**(5), 413–425.

Meijer, Erik. (1992). *Calculating compilers*. Ph.D. thesis, Nijmegen University.

Meijer, Erik, & Hutton, Graham. (1995). Bananas in space: Extending fold and unfold to exponential types. *Proc. 7th International Conference on Functional Programming and Computer Architecture*. ACM Press, San Diego, California.

Meijer, Erik, & Jeuring, Johan. (1995). Merging monads and folds for functional programming. Jeuring, Johan, & Meijer, Erik (eds), *Advanced Functional Programming*. LCNS, vol. 925. Springer-Verlag.

Meijer, Erik, Fokkinga, Maarten, & Paterson, Ross. (1991). Functional programming with bananas, lenses, envelopes and barbed wire. Hughes, John (ed), *Proc. Conference on Functional Programming and Computer Architecture*. LNCS, no. 523. Springer-Verlag.

Peterson, John, *et al.* . 1997 (Apr.). *The Haskell language report, version 1.4*. Available on the World-Wide-Web from `http://www.haskell.org`.

Reynolds, John C. (1985). Three approaches to type structure. *Proc. International Joint Conference on Theory and Practice of Software Development*. Lecture Notes in Computer Science, vol. 185. Springer.

Sheard, Tim, & Fegaras, Leonidas. (1993). A fold for all seasons. *Proc. ACM Conference on Functional Programming and Computer Architecture*. Springer.

Swierstra, S. Doaitse, Alcocer, Pablo R. Azero, & Saraiva, Joao. 1998 (Sept.). Designing and implementing combinator languages. *Lecture Notes of the 3rd International Summer School on Advanced Functional Programming*.

Takano, Akihiko, & Meijer, Erik. (1995). Shortcut deforestation in calculational form. *Proc. 7th International Conference on Functional Programming and Computer Architecture*. ACM Press, San Diego, California.

Wadler, Philip. 1981 (Oct.). Applicative style programming, program transformation, and list operators. *Proc. ACM Conference on Functional Programming Languages and Computer Architecture*.

Wadler, Philip. (1990). Comprehending monads. *Proc. ACM Conference on Lisp and Functional Programming*.

Wadler, Philip. (1992a). The essence of functional programming. *Proc. Principles of Programming Languages*.

Wadler, Philip. (1992b). Monads for functional programming. Broy, Manfred (ed), *Proc. Marktoberdorf Summer School on Program Design Calculi*. Springer–Verlag.