Views:
A way for pattern matching to cohabit with data abstraction

Philip Wadler
Programming Research Group, Oxford University, UK
and Programming Methodology Group, Chalmers University, Sweden

January 1987
(revised, March 1987)*

Abstract

Pattern matching and data abstraction are important concepts in designing programs, but they do not fit well together. Pattern matching depends on making public a free data type representation, while data abstraction depends on hiding the representation. This paper proposes the *views* mechanism as a means of reconciling this conflict. A view allows any type to be viewed as a free data type, thus combining the clarity of pattern matching with the efficiency of data abstraction.

1 Introduction

Induction and abstraction are fundamental tools of the mathematician's trade, and equally essential to the computer scientist. Pattern matching is a language feature that supports induction, and data abstraction is a feature that supports abstraction; but unfortunately these two features do not get on well together. This paper proposes the views mechanism as a means of resolving this problem.

As an example of the conflict between pattern matching and data abstraction, consider the definition of exponentiation. Mathematicians traditionally define it as follows:

$$\begin{aligned} x^0 &= 1 \\ x^{n+1} &= x(x^n) \end{aligned} \tag{1}$$

*This is a revised version of a paper presented at the 14'th ACM Symposium on Principles of Programming Languages, Munich, January 1987.

This definition makes it easy to prove properties of exponentiation by means of induction. Functional programming languages encourage a similar style of definition. For example, in a language like Hope [BMS80] or Miranda[1] [Tur85] we might declare a new type

$$peano \quad ::= \quad Zero \mid Succ\ peano$$

and then we can write a definition that is essentially equivalent to (1):

$$\begin{array}{lcl} power\ x\ Zero & = & 1 \\ power\ x\ (Succ\ n) & = & x \times power\ x\ n \end{array} \tag{2}$$

This style of definition has several advantages: each case is displayed clearly as a pattern on the left-hand side of an equation; the compiler can check that no cases have been accidentally omitted; and the definitions are well-suited for proofs by structural induction [Bur69] and for program transformation [BD77].

However, there is a problem with the above definition: it specifies a particular way of representing natural numbers, as the free data type *peano*. The representation of the number seven is the data structure

$$Succ\ (Succ\ (Succ\ (Succ\ (Succ\ (Succ\ (Succ\ Zero))))))$$

Compared with the representation of integers built-in to the computer hardware, this representation is astonishingly inefficient. If we had followed the fundamental principle of *data abstraction* (or *representation hiding*) then this problem would not arise, because we would be free to implement natural numbers in any convenient way, including the built-in integer data type.

In short, pattern matching supports clear definitions and induction, but it requires that the representing type be a free data type and be visible. Data abstraction supports efficiency, but it requires that the representing type be hidden. Thus, the programmer is often faced with an unenviable choice between clarity and efficiency.

The programming language Hope finesses this problem in the special case of the natural numbers. It provides a special mechanism that allows the built-in integer type to be viewed as if it were the type *peano*. Such a useful mechanism is clearly a candidate for generalization, as the need to view one data type as if it were another is hardly limited to this special case. This paper proposes a language mechanism called *views* as a means of satisfying this need. A view specifies how any arbitrary data type (including an abstract data type) can be viewed as a free data type. It is even possible to specify several different views of the same type.

This paper discusses views in the context of functional languages; similar ideas may be useful in imperative languages. The notation used in this paper is styled after Miranda. The essential requirement is that the host language permits the declaration of free data types. Views are useful regardless of whether eager, applicative order evaluation (as in Hope) or lazy, normal order evaluation (as in Miranda) is used; the examples in this paper work with either evaluation order.

[1] Miranda is a trademark of Research Software Limited.

Views as described here should not be confused with views in OBJ2 [FGJM85]. Views in OBJ2 specify homomorphisms between modules; views as described here specify isomorphisms between data types. (Joseph Goguen has suggested that the views in this paper be called "bi-views".)

The problem of extending pattern-matching to apply to non-free types is also addressed by Miranda, through the mechanism of "lawful types" [Tho86]. This mechanism is rather more limited than views. A lawful type is simply a subset of a free type, whereas views allow one to specify a correspondence between a free type and *any* desired type.

The remainder of this paper is organized as follows. Section 2 introduces views by showing how to define a view of integers. Section 3 briefly describes an alternative view of integers. Section 4 gives a simple example of views and abstract data types, in the context of two representations of complex numbers. Sections 5 through 8 demonstrate further applications of views to lists and trees. Section 9 shows two unusual uses of views. Section 10 and 11 describe how views support equational reasoning and induction. Section 12 outlines an efficient implementation method. Section 13 concludes.

2 Viewing an integer as zero or a successor

This section introduces the views mechanism by defining a view of the built-in integer type, *int*, that is analogous to the free data type *peano* discussed in the introduction.

Here is the definition of the view:

$$\begin{array}{lll}
\multicolumn{3}{l}{\textbf{view}\ \ int ::= Zero \mid Succ\ int} \\
\quad \textbf{in}\ n & = & Zero, \qquad\qquad \textbf{if}\ \ n = 0 \\
 & = & Succ\ (n-1), \quad \textbf{if}\ \ n > 0 \\
\quad \textbf{out}\ Zero & = & 0 \\
\quad \textbf{out}\ (Succ\ n) & = & n + 1
\end{array}$$

The first line introduces two new names, *Zero* and *Succ*, which may appear in terms (on the right-hand side of equations) and in patterns (on the left-hand side of equations).

The **in** and **out** clauses are similar to function definitions. The **in** clause defines a function to apply to an *int* to get a *Zero* or *Succ*; it is used when *Zero* or *Succ* appear in a pattern on the left-hand side of an equation. The **out** clause defines a function to apply to a *Zero* or *Succ* to get an *Int*; it is used when *Zero* or *Succ* appear on the right-hand side of an equation.

A view is well-defined only when the functions defined by the **in** and **out** clause are inverses of each other. Together, they specify an isomorphism between (a subset of) the *viewed* type and (a subset of) the *viewing* type. In this case, the isomorphism is between the natural numbers (a subset of the viewed type, *int*) and values constructed with *Succ* and *Zero* (the viewing type).

Given the above view declaration, one may write definitions such as (2) in Section 1, or the following definition of Fibonacci numbers:

$$\begin{array}{lcl} \mathit{fib}\ \mathit{Zero} & = & \mathit{Zero} \\ \mathit{fib}\ (\mathit{Succ}\ \mathit{Zero}) & = & \mathit{Succ}\ \mathit{Zero} \\ \mathit{fib}\ (\mathit{Succ}\ (\mathit{Succ}\ n)) & = & (\mathit{fib}\ n) + (\mathit{fib}\ (\mathit{Succ}\ n)) \end{array}$$

Here, views are used on the right-hand side only for symmetry (and purposes of demonstration). It would work just as well to say $\mathit{fib}\ \mathit{Zero} = 0$ for the first equation. Later, we will see examples where the use of views on the right-hand side is more natural.

This view applies only to natural numbers. Any attempt to view a negative integer as a *Zero* or *Succ* (for example, by evaluating $\mathit{fib}\ (-1)$) will cause a run-time error.

It is easy to translate a program that uses views into a program that does not use views. The view definition above is equivalent to a type definition and two function definitions:

$$\begin{array}{lcll} \multicolumn{4}{l}{\mathit{viewtype} ::= \mathit{Zero} \mid \mathit{Succ}\ \mathit{int}} \\ \mathit{viewin}\ n & = & \mathit{Zero}, & \textbf{if}\ \ n = 0 \\ & = & \mathit{Succ}\ (n-1), & \textbf{if}\ \ n > 0 \\ \mathit{viewout}\ \mathit{Zero} & = & 0 & \\ \mathit{viewout}\ (\mathit{Succ}\ n) & = & n+1 & \end{array}$$

The function *viewin* has type $\mathit{int} \rightarrow \mathit{viewtype}$ and the function *viewout* has type $\mathit{viewtype} \rightarrow \mathit{int}$.

A function definition such as *power* or *fib* is translated in two steps. First, all pattern matching is translated into case expressions; see [Aug85, Wad87]. Second, calls of *viewin* and *viewout* are inserted at appropriate places. For example, the *fib* definition above is equivalent to:

$$\begin{array}{l} \mathit{fib}\ m = \\ \quad \textbf{case}\ \ \mathit{viewin}\ m\ \ \textbf{of} \\ \quad\quad \begin{array}{lcl} \mathit{Zero} & \Rightarrow & \mathit{viewout}\ \mathit{Zero} \\ \mathit{Succ}\ m' & \Rightarrow & \textbf{case}\ \ \mathit{viewin}\ m'\ \ \textbf{of} \\ & & \quad \begin{array}{lcl} \mathit{Zero} & \Rightarrow & \mathit{viewout}\ (\mathit{Succ}\ (\mathit{viewout}\ \mathit{Zero})) \\ \mathit{Succ}\ n & \Rightarrow & \mathit{fib}\ n + \mathit{fib}\ (\mathit{viewout}\ (\mathit{Succ}\ n)) \end{array} \end{array} \end{array}$$

Note that values of type *viewtype* appear in the program only in a very restricted way (namely, as a result of *viewin* or as an argument to *viewout*).

Any view may always be expanded out in the way outlined above. Thus, views do not require any significant change in the semantics of a functional language.

3 Another view of integers

It is possible to have more than one view of a data type. An alternative view of integers is as follows:

$$
\begin{array}{llll}
\multicolumn{4}{l}{\textbf{view}\ \ \mathit{int} ::= \mathit{Zero} \mid \mathit{Even}\ \mathit{int} \mid \mathit{Odd}\ \mathit{int}} \\
\quad \textbf{in}\ n & = & \mathit{Zero}, & \textbf{if}\ \ n = 0 \\
 & = & \mathit{Even}\ (n\ \textbf{div}\ 2), & \textbf{if}\ \ n > 0 \wedge n\ \textbf{mod}\ 2 = 0 \\
 & = & \mathit{Odd}\ ((n-1)\ \textbf{div}\ 2), & \textbf{if}\ \ n > 0 \wedge n\ \textbf{mod}\ 2 = 1 \\
\quad \textbf{out}\ \mathit{Zero} & = & 0 & \\
\quad \textbf{out}\ (\mathit{Even}\ n) & = & 2 \times n, & \textbf{if}\ \ 2 \times n > 0 \\
\quad \textbf{out}\ (\mathit{Odd}\ n) & = & 2 \times n + 1, & \textbf{if}\ \ 2 \times n + 1 > 0
\end{array}
$$

The **in** and **out** clauses again define inverse functions. Note that the constructor *Zero* appears in both views; this is permissible and unambiguous because it is given the same definition in each.

Using this view one can give a more efficient definition of exponentiation:

$$
\begin{array}{lll}
\mathit{power}\ x\ \mathit{Zero} & = & 1 \\
\mathit{power}\ x\ (\mathit{Even}\ n) & = & \mathit{power}\ (x \times x)\ n \\
\mathit{power}\ x\ (\mathit{Odd}\ n) & = & x \times \mathit{power}\ (x \times x)\ n
\end{array}
$$

This expresses the traditional divide-and-conquer algorithm.

4 Viewing a complex number in cartesian and polar coordinates

This section gives a simple illustration of how abstract data types can be combined with pattern matching. The pattern matching here is extremely simple—no case analysis or recursive types are involved—but still useful.

Two well known representations of complex numbers are the polar and the cartesian. We might choose to represent complex numbers in the polar representation, and provide the cartesian as a view:

$$\mathit{complex} ::= \mathit{Pole}\ \mathit{real}\ \mathit{real}$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{view}\ \ \mathit{complex} ::= \mathit{Cart}\ \mathit{real}\ \mathit{real}} \\
\quad \textbf{in}\ (\mathit{Pole}\ r\ t) & = & \mathit{Cart}\ (r \times \mathit{cos}\ t)\ (r \times \mathit{sin}\ t), \\
 & & \quad \textbf{if}\ \ (0 < r \wedge 0 \leq t \wedge t < 2 \times \mathit{pi}) \vee (0 = r \wedge 0 = t) \\
\quad \textbf{out}\ (\mathit{Cart}\ x\ y) & = & \mathit{Pole}\ (\mathit{sqrt}\ (x^2 + y^2))\ (\mathit{atan2}\ x\ y)
\end{array}
$$

The constraints on r and t are necessary to ensure that there is an isomorphism between the polar and cartesian representations of points; otherwise, many polar representations might correspond to one cartesian representation. The function *atan2 x y* returns the arctangent of x/y properly adjusted for quadrant, and returns a sensible value if y is zero (0 if $x \geq 0$, and π if $x < 0$).

Multiplication and addition of complex numbers can now be defined as follows:

$$\begin{array}{lcl} \mathit{add}\ (\mathit{Cart}\ x\ y)\ (\mathit{Cart}\ x'\ y') & = & \mathit{Cart}\ (x + x')\ (y + y') \\ \mathit{mult}\ (\mathit{Pole}\ r\ t)\ (\mathit{Pole}\ r'\ t') & = & \mathit{Pole}\ (r \times r')\ (t + t') \end{array}$$

Here addition was defined in terms of the cartesian representation and multiplication in terms of the polar representation.

Alternatively, we might choose to represent complex numbers in the cartesian representation, and provide the polar as a view. This requires just a small variation on the previous declaration:

$$\mathit{complex} ::= \mathit{Cart}\ \mathit{real}\ \mathit{real}$$

$$\begin{array}{l} \textbf{view}\ \ \mathit{complex} ::= \mathit{Pole}\ \mathit{real}\ \mathit{real} \\ \quad \begin{array}{lcl} \textbf{in}\ (\mathit{Cart}\ x\ y) & = & \mathit{Pole}\ (\mathit{sqrt}\ (x^2 + y^2))\ (\mathit{atan2}\ x\ y), \\ \textbf{out}\ (\mathit{Pole}\ r\ t) & = & \mathit{Cart}\ (r \times \mathit{cos}\ t)\ (r \times \mathit{sin}\ t), \\ & & \quad \textbf{if}\ \ (0 < r \wedge 0 \leq t \wedge t < 2 \times \mathit{pi}) \vee (0 = r \wedge 0 = t) \end{array} \end{array}$$

Here the **in** and **out** clauses are reversed from their previous positions. The definitions of *add* and *mult* given previously are still valid under this new declaration. This shows how views can be used to hide choice of representation, while still allowing the convenience of pattern matching.

The traditional method for hiding a representation is an abstract data type. Just as abstract data types export values and functions, they should also be able to export views. It is easy to modify the abstract data type mechanism of Miranda to include views, for example:

$$\begin{array}{l} \textbf{abstype}\ \ \mathit{complex}\ \ \textbf{with} \\ \quad \mathit{complex} ::= \mathit{Cart}\ \mathit{real}\ \mathit{real} \\ \quad \mathit{complex} ::= \mathit{Pole}\ \mathit{real}\ \mathit{real} \end{array}$$

This abstract type only defines views, but in general an abstract type might define both views and functions; the syntax for declaring functions is shown below.

This declaration can be implemented with a polar representation and a cartesian view, or with a cartesian representation and a polar view, or with some third representation and cartesian and polar both as views. The definitions of *add* and *mult* will still be valid regardless of the representation chosen.

For comparison, consider the equivalent abstract type if views are not used:

$$\begin{array}{l} \textbf{abstype}\ \ \mathit{complex}\ \ \textbf{with} \\ \quad \mathit{xpart}, \mathit{ypart}, \mathit{rpart}, \mathit{tpart} :: \mathit{complex} \rightarrow \mathit{real} \\ \quad \mathit{mkcart}, \mathit{mkpole} :: \mathit{real} \rightarrow \mathit{real} \rightarrow \mathit{complex} \end{array}$$

The single identifier *Cart* has been replaced by three identifiers, *xpart*, *ypart*, and *mkcart*; and similarly for *Pole*. The corresponding definitions of addition and multiplication are:

$$\begin{array}{lcl} \mathit{add}\ c\ c' & = & \mathit{mkcart}\ (\mathit{xpart}\ c + \mathit{xpart}\ c')\ (\mathit{ypart}\ c + \mathit{ypart}\ c') \\ \mathit{mult}\ c\ c' & = & \mathit{mkpole}\ (\mathit{rpart}\ c \times \mathit{rpart}\ c')\ (\mathit{tpart}\ c + \mathit{tpart}\ c') \end{array}$$

In this example the difference is not great, but for more complicated examples the advantages of views—compactness and the ability to use pattern matching—would be more pronounced.

The example given here is slightly contrived, because in practice it would be more sensible to define the operations *add* and *mult* inside the abstraction, rather than outside it. However, it serves to show how views combine well with data abstraction, and particularly how views can reconcile the conflict between pattern matching and data abstraction mentioned in the introduction. One way to summarize this result is as follows. Traditionally, abstraction is achieved by refusing to export the representation. With views, abstraction can be achieved by exporting as many representations as desired.

5 Viewing a list backwards

Assume that lists are represented in the traditional way, so that, for example, $[1, 2]$ is taken as an abreviation for 1 **Cons** $(2$ **Cons** $\mathit{Nil})$, where the constructors *Nil* and **Cons** are defined as follows:

$$\mathit{list}\ \alpha \quad ::= \quad \mathit{Nil} \mid \alpha\ \mathbf{Cons}\ (\mathit{list}\ \alpha)$$

(Here x **Cons** xs is just different syntax for $\mathit{Cons}\ x\ xs$.)

Of course, the **Cons** representation of lists is “biased” towards the first element of the list. For example, it is much easier to write a function to return the first element of the list than a function to return the last:

$$\begin{array}{lcl} \mathit{head}\ (x\ \mathbf{Cons}\ xs) & = & x \\ \\ \mathit{last}\ (x\ \mathbf{Cons}\ \mathit{Nil}) & = & x \\ \mathit{last}\ (x\ \mathbf{Cons}\ (x'\ \mathbf{Cons}\ xs)) & = & \mathit{last}\ (x'\ \mathbf{Cons}\ xs) \end{array}$$

We can define a new view, based on constructors *Nil* and **Snoc**, that is biased in the opposite way:

$$\begin{array}{l} \mathbf{view}\ \ \mathit{list}\ \alpha ::= \mathit{Nil} \mid (\mathit{list}\ \alpha)\ \mathbf{Snoc}\ \alpha \\ \quad \begin{array}{lcl} \mathbf{in}\ (x\ \mathbf{Cons}\ \mathit{Nil}) & = & \mathit{Nil}\ \mathbf{Snoc}\ x \\ \mathbf{in}\ (x\ \mathbf{Cons}\ (xs\ \mathbf{Snoc}\ x')) & = & (x\ \mathbf{Cons}\ xs)\ \mathbf{Snoc}\ x' \\ \mathbf{out}\ (\mathit{Nil}\ \mathbf{Snoc}\ x) & = & x\ \mathbf{Cons}\ \mathit{Nil} \\ \mathbf{out}\ ((x\ \mathbf{Cons}\ xs)\ \mathbf{Snoc}\ x') & = & x\ \mathbf{Cons}\ (xs\ \mathbf{Snoc}\ x') \end{array} \end{array}$$

This allows us to view the list $[1, 2]$ as if were $(\mathit{Nil}$ **Snoc** $1)$ **Snoc** 2. We can now write definitions such as

$$\begin{array}{lcl} \mathit{last}\ (xs\ \mathbf{Snoc}\ x) & = & x \\ \mathit{rotleft}\ (x\ \mathbf{Cons}\ xs) & = & xs\ \mathbf{Snoc}\ x \\ \mathit{rotright}\ (xs\ \mathbf{Snoc}\ x) & = & x\ \mathbf{Cons}\ xs \end{array}$$

Here *last* is equivalent to the definition above, and, for example, $\mathit{rotleft}\ [1, 2, 3, 4] = [2, 3, 4, 1]$ and $\mathit{rotright}\ [1, 2, 3, 4] = [4, 1, 2, 3]$.

Attention is drawn to three features of the above view.

First, the constructor *Nil*, which is part of the representation, also appears in the view, just as *Zero* was shared between the two different views of integers. In general, it is permissible (and unambiguous) to let views of the same representation share any number of constructors with the representation and with each other.

Second, some left-hand sides in the **in** clause above contain **Snoc**. Matching against these will cause a recursive invocation of **in**. The **out** clause is recursive in a similar way. Recursive **in** and **out** clauses are perfectly acceptable, in the same way that recursive function definitions are.

Third, in the definition above the **in** and **out** clauses are exact inverses of one another. This can be abbreviated as follows:

$$\begin{array}{lcl} \textbf{view}\ \ \mathit{list}\ \alpha ::= \mathit{Nil} \mid (\mathit{list}\ \alpha)\ \textbf{Snoc}\ \alpha & & \\ \quad \textbf{inout}\ (x\ \textbf{Cons}\ \mathit{Nil}) & = & \mathit{Nil}\ \textbf{Snoc}\ x \\ \quad \textbf{inout}\ (x\ \textbf{Cons}\ (xs\ \textbf{Snoc}\ x')) & = & (x\ \textbf{Cons}\ xs)\ \textbf{Snoc}\ x' \end{array}$$

As mentioned previously, a view is well-defined only when the **in** and **out** clauses define functions that are inverses of one another. In the case that a view can be defined using an **inout** clause, this property follows automatically.

Of course, the representation is still biased. For example, x **Cons** xs evaluates much more efficiently than xs **Snoc** x. Also, consider the two function definitions,

$$\begin{array}{l} f_1\ (x\ \textbf{Cons}\ (x'\ \textbf{Cons}\ xs)) = e_1\ x\ x'\ xs \\ f_2\ ((xs\ \textbf{Snoc}\ x')\ \textbf{Snoc}\ x) = e_2\ xs\ x'\ x \end{array}$$

The matching in f_1 takes constant time, while the matching in f_2 takes time proportional to the length of the input list. Indeed, the input list is traversed once to decompose the list for the outer **Snoc**, and then the entire input list (except for its last element) is traversed again to decompose it for the inner **Snoc**. The next section descibes a representation where all bias is removed, and the cons and snoc views are equally efficient.

6 The join representation of lists

As an alternative to the cons representation of lists, several researchers have suggested the following representation [Mee84, SH82]:

$$\mathit{list}\ \alpha ::= \mathit{Nil} \mid \mathit{Unit}\ \alpha \mid (\mathit{list}\ \alpha)\ \textbf{Join}\ (\mathit{list}\ \alpha)$$

(Interestingly, Meertens has made this suggestion for reasons of mathematical elegance, whereas Sleep and Holmström have suggested it for reasons of efficiency!) Each list now has many possible representations. For example, the list $[1, 2]$ might be represented by any of the following:

$$\begin{array}{c} (\mathit{Unit}\ 1)\ \textbf{Join}\ (\mathit{Unit}\ 2) \\ (\mathit{Nil}\ \textbf{Join}\ (\mathit{Unit}\ 1))\ \textbf{Join}\ ((\mathit{Unit}\ 2)\ \textbf{Join}\ \mathit{Nil}) \\ (\mathit{Unit}\ 1)\ \textbf{Join}\ ((\mathit{Unit}\ 2)\ \textbf{Join}\ \mathit{Nil}) \end{array}$$

Indeed, each list has a potentially infinite number of representations, since xs and Nil **Join** xs both represent the same list.

Assuming this new representation, we can define a view that allows one to view a join list as if it were a cons list:

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\textbf{view}\ \ list\ \alpha ::= Nil \mid \alpha\ \textbf{Cons}\ (list\ \alpha)} \\
\quad \textbf{in}\ (Unit\ x) & = & x\ \textbf{Cons}\ Nil \\
\quad \textbf{in}\ (Nil\ \textbf{Join}\ xs) & = & \textbf{in}\ xs \\
\quad \textbf{in}\ ((Unit\ x)\ \textbf{Join}\ xs) & = & x\ \textbf{Cons}\ xs \\
\quad \textbf{in}\ ((xs\ \textbf{Join}\ ys)\ \textbf{Join}\ zs) & = & \textbf{in}\ (xs\ \textbf{Join}\ (ys\ \textbf{Join}\ zs)) \\
\quad \textbf{out}\ (x\ \textbf{Cons}\ xs) & = & (Unit\ x)\ \textbf{Join}\ xs
\end{array}
$$

The **in** clause maps all of the different ways of representing $[1, 2]$ as a join list into the same view as a cons list, namely

$$1\ \textbf{Cons}\ (2\ \textbf{Cons}\ Nil).$$

Conversely, the **out** clause maps this term back into a particular representation as a join list, namely

$$(Unit\ 1)\ \textbf{Join}\ ((Unit\ 2)\ \textbf{Join}\ Nil).$$

The correctness of the view depends on the equivalence between the various ways of representing a join list; otherwise, the **in** and **out** functions would not be inverses.

Note that recursion in the **in** clause above is indicated explicitly, as compared with the implicit recursion in the snoc view of the previous section.

The equation in the **out** clause above also appears, inverted, in the **in** clause. Thus, it is easy to show that the **in** and **out** clauses define inverse functions. In this case, the **out** clause cannot be omitted in favour of an **inout** clause, because a cons list can be represented in more than one way by a join list. Choosing which equations to include in the **out** clause is equivalent to choosing which representation to use.

A snoc view of join lists can be defined in a way completely symmetric to the definition given above. This is left as an exercise for the reader.

A join list can always be viewed as a cons list or a snoc list in time proportional to the size of the join list. (Further, unless the join list has an abundance of Nil nodes, its size will be proportional to the size of the corresponding cons and snoc lists.) Also, when the join representation is used, appending two lists requires constant time, rather than time proportional to the size of one of the lists. It is these properties that make the join representation desirable in terms of efficiency.

7 Viewing a list of pairs as a pair of lists

The function that converts a pair of lists into a list of pairs is defined as follows:

$$
\begin{array}{lcl}
zip\ (Nil, Nil) & = & Nil \\
zip\ (a\ \textbf{Cons}\ as, b\ \textbf{Cons}\ bs) & = & (a, b)\ \textbf{Cons}\ zip\ (as, bs)
\end{array}
$$

(The pair notation is just another syntax for constructors; think of (a, b) as equivalent to $Pair\ a\ b$.) For example,

$$zip\ ([1, 2, 3], [\text{'a'}, \text{'b'}, \text{'c'}]) = [(1, \text{'a'}), (2, \text{'b'}), (3, \text{'c'})]$$

Very often, it is also necessary to decompose a list of pairs into a pair of lists. This is usually done by an idiom such as the following:

$$\begin{array}{lll} f\ cs & = & e\ as\ bs \\ & & \quad \textbf{where}\ as\ =\ [a \mid (a, b) \leftarrow cs] \\ & & \quad \phantom{\textbf{where}}\ bs\ =\ [b \mid (a, b) \leftarrow cs] \end{array}$$

(This uses *list comprehension* notation; see [Tur81, Wad87].)

Clearly, *zip* defines a isomorphism: given a list of pairs *cs* there is always a unique pair of lists *as* and *bs* such that $zip\ (as, bs) = cs$. Thus, we can discard the definition of *zip* given above, and instead define a view *Zip* of pairs of lists:

$$\begin{array}{ll} \multicolumn{2}{l}{\textbf{view}\ \ list\ (\alpha, \beta) ::= Zip\ (list\ \alpha, list\ \beta)} \\ \quad \textbf{inout}\ Nil & = \ Zip\ (Nil, Nil) \\ \quad \textbf{inout}\ ((a, b)\ \textbf{Cons}\ Zip\ (as, bs)) & = \ Zip\ (a\ \textbf{Cons}\ as, b\ \textbf{Cons}\ bs) \end{array}$$

(Here the type of the pair (a, b) is written (α, β).) Now we can write $Zip\ (as, bs)$ in place of $zip\ (as, bs)$ on the right-hand side of equations. We can also write $Zip\ (as, bs)$ on the left-hand sides of equations. For example, the idiom given above becomes

$$f\ (Zip\ (as, bs)) \ = \ e\ as\ bs$$

which is somewhat shorter.

8 Two representations of trees

One common representation of trees is the following:

$$tree\ \alpha \ ::= \ Leaf\ \alpha \mid Branch\ (tree\ \alpha)\ (tree\ \alpha)$$

Another common representation uses what is called the "spine" (mixing an anatomical metaphor with an arboreal one). Spine trees, together with the isomorphism that relates them to branch trees, can be conveniently described by the following view:

$$\begin{array}{ll} \multicolumn{2}{l}{\textbf{view}\ \ tree\ \alpha ::= Spine\ \alpha\ (list\ (tree\ \alpha))} \\ \quad \textbf{inout}\ (Leaf\ x) & = \ Spine\ x\ Nil \\ \quad \textbf{inout}\ (Branch\ (Spine\ x\ xts)\ xt) & = \ Spine\ x\ (xts\ \textbf{Snoc}\ xt) \end{array}$$

For example, the branch tree

$$Branch\ (Branch\ (Leaf\ \text{"f"})\ (Leaf\ \text{"a"}))\ (Leaf\ \text{"b"})$$

is equivalent to the spine tree

$$Spine \text{ "f" } [Spine \text{ "a" } [\,], Spine \text{ "b" } [\,]].$$

Of course, one could use spine trees as the underlying representation with branch trees as the view, or hide the representation in an abstract type and provide both branch and spine as views.

9 Two other uses of views

This section presents two rather unusual uses of views. It is not clear whether these uses should be considered good style, but they do demonstrate the power of the view mechanism.

Occasionally, it is convenient to both match an argument against a pattern, and to refer to it by a single name. (One might say we wish to "eat our argument and have it too".) Hope provides the **as** construct for this purpose. For example, one might write:

$$\begin{array}{lcl} \mathit{factorial}\ (n\ \mathbf{as}\ \mathit{Zero}) & = & 1 \\ \mathit{factorial}\ (n\ \mathbf{as}\ \mathit{Succ}\ n') & = & n \times \mathit{factorial}\ n' \end{array}$$

Surprisingly, **as** can be defined as a view:

$$\begin{array}{lcll} \multicolumn{4}{l}{\mathbf{view}\ \ \alpha ::= \alpha\ \mathbf{as}\ \alpha} \\ \quad \mathbf{in}\ x & = & x\ \mathbf{as}\ x & \\ \quad \mathbf{out}\ (x\ \mathbf{as}\ x') & = & x, & \mathbf{if}\ \ x = x' \end{array}$$

This defines an isomorphism between any type α, and the subset of the viewing type α **as** α where the left and right arguments of the constructor are equal. It is unlikely that one would want to use **as** on the right-hand side in an equation, but the **out** clause is necessary for the view to be well-defined.

One may even use views in place of predicates. For example, one might define:

$$\begin{array}{lcll} \multicolumn{4}{l}{\mathbf{view}\ \ \mathit{int} ::= \mathit{EvenP}\ \mathit{int} \mid \mathit{OddP}\ \mathit{int}} \\ \quad \mathbf{in}\ n & = & \mathit{EvenP}\ n, & \mathbf{if}\ \ n\ \mathbf{mod}\ 2 = 0 \\ & = & \mathit{OddP}\ n, & \mathbf{if}\ \ n\ \mathbf{mod}\ 2 = 1 \\ \quad \mathbf{out}\ \mathit{EvenP}\ n & = & n, & \mathbf{if}\ \ n\ \mathbf{mod}\ 2 = 0 \\ \quad \mathbf{out}\ \mathit{OddP}\ n & = & n, & \mathbf{if}\ \ n\ \mathbf{mod}\ 2 = 1 \end{array}$$

Then we can write

$$\begin{array}{lcl} f\ (\mathit{EvenP}\ n) & = & e_1\ n \\ f\ (\mathit{OddP}\ n) & = & e_2\ n \end{array}$$

instead of

$$\begin{array}{lcll} f\ n & = & e_1\ n, & \mathbf{if}\ \ n\ \mathbf{mod}\ 2 = 0 \\ & = & e_2\ n, & \mathbf{if}\ \ n\ \mathbf{mod}\ 2 = 1 \end{array}$$

Replacing conditions by patterns may occasionally be clearer, particularly if many functions test the same condition. It may also improve clarity when a function has many arguments, and the test of the condition interacts with pattern matching for the other arguments. From the **out** clause, it can be seen that the term $\mathit{EvenP}\ n$ is equivalent to n with the additional assertion that n is even. Conceivably, it might be useful to use *EvenP* and *OddP* on the right-hand sides of equations, as a way of documenting that certain conditions hold.

10 Equational reasoning

In order for a language feature to be useful, it must be easy to reason about programs containing that feature. Views have been carefully designed to support two important proof techniques, equational reasoning and induction. These are discussed in this section and the next.

Equational reasoning is a principle of such supreme importance that it goes by many names: referential transparency, the rule of Leibniz, and more plainly "substituting equals for equals". As a very simple example, given the function definition

$$\begin{aligned} \mathit{last}\ (x\ \mathbf{Cons}\ \mathit{Nil}) &= x & (1)\\ \mathit{last}\ (x\ \mathbf{Cons}\ (x'\ \mathbf{Cons}\ xs)) &= \mathit{last}\ (x'\ \mathbf{Cons}\ xs) & (2) \end{aligned}$$

equational reasoning is sufficient to calculate the value of last ['a', 'b'], as follows:

$$\begin{aligned} \mathit{last}\ (\text{`a'}\ \mathbf{Cons}\ (\text{`b'}\ \mathbf{Cons}\ \mathit{Nil})) &= \mathit{last}\ (\text{`b'}\ \mathbf{Cons}\ \mathit{Nil}) & \text{(by 2)}\\ &= \text{`b'} & \text{(by 1)} \end{aligned}$$

A key principle in the design of views is that all equations in the **in** and **out** clauses of the view can be used just like any other equations for equational reasoning. Thus, given the view

$$\begin{aligned} &\mathbf{view}\ \ \mathit{list}\ \alpha ::= \mathit{Nil} \mid (\mathit{list}\ \alpha)\ \mathbf{Snoc}\ \alpha\\ &\quad \mathbf{inout}\ (x\ \mathbf{Cons}\ \mathit{Nil}) = \mathit{Nil}\ \mathbf{Snoc}\ x & (3)\\ &\quad \mathbf{inout}\ (x\ \mathbf{Cons}\ (xs\ \mathbf{Snoc}\ x')) = (x\ \mathbf{Cons}\ xs)\ \mathbf{Snoc}\ x' & (4) \end{aligned}$$

and the definition

$$\mathit{last}\ (xs\ \mathbf{Snoc}\ x) = x \qquad (5)$$

we may now calculate as follows:

$$\begin{aligned} &\mathit{last}\ (\text{`a'}\ \mathbf{Cons}\ (\text{`b'}\ \mathbf{Cons}\ \mathit{Nil}))\\ &\quad = \mathit{last}\ (\text{`a'}\ \mathbf{Cons}\ (\mathit{Nil}\ \mathbf{Snoc}\ \text{`b'})) & \text{(by 3)}\\ &\quad = \mathit{last}\ ((\text{`a'}\ \mathbf{Cons}\ \mathit{Nil})\ \mathbf{Snoc}\ \text{`b'}) & \text{(by 4)}\\ &\quad = b & \text{(by 5)} \end{aligned}$$

Of course, the main value of equational reasoning is not in calculating values but in performing proofs. Given the definitions

$$\begin{array}{rcll} \mathit{rotleft}\ (x\ \mathbf{Cons}\ xs) & = & xs\ \mathbf{Snoc}\ x & (6) \\ \mathit{rotright}\ (xs\ \mathbf{Snoc}\ x) & = & x\ \mathbf{Cons}\ xs & (7) \end{array}$$

we may prove

$$\mathit{rotright}\ (\mathit{rotleft}\ xs) = xs$$

for a non-empty finite list xs, by simply observing that

$$\begin{array}{rcll} \mathit{rotright}\ (\mathit{rotleft}\ (x\ \mathbf{Cons}\ xs) & = & \mathit{rotright}\ (xs\ \mathbf{Snoc}\ x) & (\text{by } 6) \\ & = & x\ \mathbf{Cons}\ xs & (\text{by } 7) \end{array}$$

Both the definitions and the proof are rather more involved if views are not used.

Equational reasoning is valid for all definitions that use pattern matching over free data types. A view establishes an isomorphism between (a subset of) the viewed data type and (a subset of) a free data type. Thus, equational reasoning is also valid for all definitions that use pattern matching over views.

However, some caution is required, because the view may imply additional conditions that the program must satisfy. For example, consider the view of join lists as cons lists. as cons lists. A look at this view will reveal that it establishes the following conditions on join lists:

$$\begin{array}{rcl} \mathit{Nil}\ \mathbf{Join}\ xs & = & xs \\ xs\ \mathbf{Join}\ (ys\ \mathbf{Join}\ zs) & = & (xs\ \mathbf{Join}\ ys)\ \mathbf{Join}\ zs \end{array}$$

That is, **Join** must have *Nil* as a (left) identity and be associative. The programmer must verify that every definition containing **Join** is consistent with these conditions. It is also desirable that definitions be consistent with the condition

$$xs\ \mathbf{Join}\ \mathit{Nil} \quad = \quad xs$$

but this is not required by the view.

An example of a satisfactory definition is

$$\begin{array}{lcl} \mathit{length}\ \mathit{Nil} & = & 0 \\ \mathit{length}\ (\mathit{Unit}\ x) & = & 1 \\ \mathit{length}\ (xs\ \mathbf{Join}\ ys) & = & \mathit{length}\ xs + \mathit{length}\ ys \end{array}$$

This definition establishes a homomorphism mapping **Join** onto $+$ and *Nil* onto 0. It is valid because $+$ has 0 as an identity and is associative, and so the desired properties are preserved. (For further discussion of homomorphisms and operations on lists, the reader is referred to [Mee84, Bir86].)

An example of an unsatisfactory definition is

$$\begin{array}{lcl} \mathit{silly}\ \mathit{Nil} & = & 1 \\ \mathit{silly}\ (\mathit{Unit}\ x) & = & 2 \\ \mathit{silly}\ (xs\ \mathbf{Join}\ ys) & = & \mathit{silly}\ xs + \mathit{silly}\ ys \end{array}$$

This definition does not preserve the desired properties, because 1 is not a right identity of $+$. Thus, from the condition that $xs = \mathit{Nil}\ \mathbf{Join}\ xs$ we could derive

$$2 = \mathit{silly}\ (\mathit{Unit}\ \text{`a'}) = \mathit{silly}\ (\mathit{Nil}\ \mathbf{Join}\ (\mathit{Unit}\ \text{`a'})) = 3$$

It is impossible to reason equationally about a program containing this definition, which is just as well since it is a silly definition indeed.

A further example of an unsatisfactory definition is

$$\mathit{nothead}\ ((\mathit{Unit}\ x)\ \mathbf{Join}\ xs) \quad = \quad x$$

This definition won't do, because we have

$$(\mathit{Unit}\ \text{`a'})\ \mathbf{Join}\ (\mathit{Unit}\ \text{`b'}) = \mathit{Nil}\ \mathbf{Join}\ ((\mathit{Unit}\ \text{`a'})\ \mathbf{Join}\ (\mathit{Unit}\ \text{`b'}))$$

but applying *nothead* to the left of this equation yields 'a', while applying *nothead* to the right yields an undefined value.

One should consider it a bonus that views reveal an additional condition that definitions involving **Join** must be checked for. On the other hand, it is useful to be able to limit the portions of the program that must be so checked. This can be done in the usual way, by encapsulating **Join** within an abstract data type. Further, it is completely safe to export the cons and snoc views of lists outside of this type, because they are guaranteed to enforce the necessary conditions. For example, the definition

$$\mathit{head}\ (x\ \mathbf{Cons}\ xs) \quad = \quad x$$

is completely satisfactory, because the cons view necessarily respects the equivalences among various join representations of the same list.

11 Induction

It is also essential that induction should work over views. For example, in order to demonstrate that a property $P(xs)$ holds for every finite list xs, it is sufficient to show

1. $P(\mathit{Nil})$ holds, and
2. $P(xs\ \mathbf{Snoc}\ x)$ holds, assuming $P(xs)$ holds.

Similarly, one can prove properties of the natural numbers by inducting over *Zero* and *Succ*.

In general, induction over a view is valid to demonstrate properties that hold for all elements *generated* by the viewing type. For example, every finite list can be generated using *Nil* and **Snoc**, so induction over these serves to prove properties of finite lists. Similarly, every natural number can be generated using *Zero* and *Succ*, so induction over these serves to prove properties of natural numbers. Here the induction only demonstrates properties of a subset (natural numbers) of the viewed type (integers).

A more problematic example is the unusual view of even and odd integers presented in Section 9. The set of values that can be generated using just the two constructors *EvenP* and *OddP* is empty, and so induction is not appropriate over this view.

12 Implementation

As explained in Section 2, each view can be expanded out into the definition of a new type, *viewtype*, and applications of conversion functions, *viewin* and *viewout*. This is workable, but not ideal. Constructors of the viewed type (such as *Zero* and *Succ*) appear only as the result of *viewin* or the argument to *viewout*. At run-time, such constructors are always allocated storage and then immediately examined and never referenced again. It would be preferable to use a scheme that avoided allocating such constructors altogether.

Such a scheme is possible. Instead of introducing a new type and two conversion functions, in the modified scheme a view with k constructors introduces $k+1$ new functions (and no new types). One function acts as the equivalent of *viewin* and the associated **case** expression, while the remaining k functions provide an equivalent of *viewout* for each constructor.

For example, the view of integers in Section 2 translates into the following three functions:

$$\begin{array}{lcll} \mathit{viewcase}\ z\ s\ n & = & z & \textbf{if}\ \ n = 0 \\ & = & s\ (n-1) & \textbf{if}\ \ n > 0 \\ \\ \mathit{zero} & = & 0 & \\ \mathit{succ}\ n & = & n+1 & \end{array}$$

Note that *viewcase* is designed so that if n corresponds to *Zero* then z is returned, and if n corresponds to $Succ\ n'$ then $s\ n'$ is returned. Since s is itself a function, *viewcase* can be defined only in a higher-order language, where functions may be passed to other functions.

Under this scheme, the definition of *fib* now translates as follows:

$$\begin{array}{lcl} \mathit{fib}\ m & = & \mathit{viewcase} \\ & & \quad \mathit{zero} \\ & & \quad (\lambda m'.\mathit{viewcase} \\ & & \qquad\qquad (\mathit{succ}\ \mathit{zero}) \\ & & \qquad\qquad (\lambda n.\ n + \mathit{fib}\ (\mathit{succ}\ n)) \\ & & \qquad\qquad m') \\ & & \quad m \end{array}$$

It is left to the reader to verify that this translation is equivalent to the one described in Section 2.

The new translation scheme has the desired property that it introduces no new constructors (like *Succ* and *Zero*) and hence requires no extra allocation operations at run-time. However, it has the disadvantage that application of *viewcase* may be more difficult to evaluate efficiently than the corresponding **case** expressions. The exact nature of this problem will depend on the evaluation method used. Here the problem will be considered in the context of a G-machine style compiler, and a solution suggested. Some familiarity with the G-machine is assumed; see [Aug84, Joh85, Pey87] for an introduction.

As an example, consider the following definition (which happens to provide a fast method for calculating Fibonacci numbers):

$$\begin{array}{lcl} \mathit{fibx}\ a\ b\ \mathit{Zero} & = & a \\ \mathit{fibx}\ a\ b\ (\mathit{Succ}\ n) & = & \mathit{fibx}\ b\ (a+b)\ n \end{array}$$

Translated using the above scheme, this becomes

$$\mathit{fibx}\ a\ b\ m \ = \ \mathit{viewcase}\ a\ (\lambda n.\mathit{fibx}\ b\ (a+b)\ n)\ m$$

After lambda-lifting, this definition in turn becomes

$$\begin{array}{lcl} \mathit{fibx}\ a\ b\ m & = & \mathit{viewcase}\ a\ (\mathit{fibx}_0\ a\ b)\ m \\ \mathit{fibx}_0\ a\ b\ n & = & \mathit{fibx}\ b\ (a+b)\ n \end{array}$$

At run-time, the code implementing the body of *fibx* will need to allocate heap storage to represent the application $(\mathit{fibx}_0\ a\ b)$. Thus, we have saved the allocation of a constructor *Zero* or *Succ* only to replace it by a potentially larger allocation (since in general storage will need to be allocated for each free variable in each argument to *viewcase*).

Fortunately, this problem can be solved by simply rearranging the way free variables are passed into arguments to *viewcase*. In the rearranged method, all free variables of the function containing the *viewcase* are lambda-abstracted from each argument to the *viewcase* (and in the same order); these variables are then passed into the result returned by the *viewcase*. Thus, the definition of *fibx* now translates to

$$\begin{array}{l} \mathit{fibx}\ a\ b\ m = \\ \quad \mathit{viewcase}\ (\lambda a.\lambda b.\lambda m.a)\ (\lambda n.\lambda a.\lambda b.\lambda m.\mathit{fibx}\ b\ (a+b)\ n)\ m\ a\ b\ m \end{array}$$

After lambda-lifting this becomes

$$\begin{array}{lcl} \mathit{fibx}\ a\ b\ m & = & \mathit{viewcase}\ \mathit{fibx}_1\ \mathit{fibx}_2\ m\ a\ b\ m \\ \mathit{fibx}_1\ a\ b\ m & = & a \\ \mathit{fibx}_2\ n\ a\ b\ m & = & \mathit{fibx}\ b\ (a+b)\ n \end{array}$$

Now the body of *fibx* need merely push pointers to fibx_1 and fibx_2 onto the stack, copy the parameter m onto the top of the stack, and perform a tail-call (that is, a jump) to *viewcase*. In turn, *viewcase* will examine the argument m; if it is zero it will perform a tail-call (jump) to fibx_1, and if it is positive it will push $m - 1$ onto the stack and perform a tail-call (jump) to fibx_2. By passing the free variables in the body of *fibx* to each argument in the same order, the rearrangement of the stack necessary at run-time is minimised. (Given the above definition of *fibx*, the current G-machine compiler would not produce quite the sequence of steps described here, but it could be modified to do so.)

This is quite acceptably efficient. Further efficiency might be gained by expanding out non-recursive function applications at compile time. (The compilers described by

[HK84, FW86] perform expansion of this kind.) For example, performing expansion on the above definition of *fibx* yields

$$\begin{aligned} \mathit{fibx}\ a\ b\ m &= a, && \textbf{if}\ \ m = 0 \\ &= \mathit{fibx}\ b\ (a + b)\ (m - 1), && \textbf{if}\ \ m > 0 \end{aligned}$$

which resembles the code one would write if a view had not been used. However, it is difficult to see how to perform an equivalent expansion of a recursive *viewcase*, such as the one associated with the snoc view of lists.

13 Conclusions

Designers of software are continually faced with trade-offs. Some of these trade-offs are necessary, but others can be avoided by careful design. It is particularly worrying when we are forced to choose between valuable methods such as pattern matching and data abstraction. Views move this trade-off from the "necessary" to the "avoidable" category.

After the views mechanism was defined, several unexpected uses of it emerged. These included the list-of-pairs to pair-of-lists view discussed in Section 7 and the two unusual views discussed in Section 9. No doubt many other uses of views are waiting to be discovered.

Programming languages are awash with features, and new features must be approached with caution. Views are worth consideration because they address an important need—reconciling pattern matching with data abstraction. In doing so, they also bring a new perspective. Instead of thinking of an abstract data type as hiding a representation, with views we can think of it as exporting as many representations as convenient.

Acknowledgements. I am grateful to Tony Hoare for pointing out how views should support equational reasoning; Joseph Goguen for providing useful comments on an earlier draft of this report; Bernard Sufrin and John Hughes for acting as sounding boards for these ideas; Sören Holmström for noting that the original definition of the cartesian/polar view did not specify an isomorphism; and Thomas Johnsson for his careful proofreading.

This research was performed while on a fellowship supported by ICL.

References

[Aug84] L. Augustsson. A compiler for lazy ML. In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 218–227, Austin, 1984.

[Aug85] L. Augustsson. Compiling pattern matching. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, Springer-Verlag, September 1985.

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

[Bir86] R. S. Bird. An introduction to the theory of lists. In *Marktoberdorf Workshop on Logics of Programming*, August 1986.

[BMS80] R. Burstall, D. MacQueen, and D. Sanella. *Hope: An experimental applicative language.* Technical Report Report CSR-62-80, Edinburgh University, Computer Science Dept., 1980.

[Bur69] R. M. Burstall. Proving properties of programs by structural induction. *The Computer Journal*, 12(1), February 1969.

[FGJM85] K. Futasagi, J. A. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In *ACM Symposium on Principles of Programming Languages*, pages 52–66, January 1985.

[FW86] J. Fairbairn and S. C. Wray. Code generation techniques for functional languages. In *Proceedings of the 1986 ACM Symposium on Lisp and Functional Programming*, pages 94–104, Boston, 1986.

[HK84] P. Hudak and D. Kranz. A combinator-based compiler for a functional language. In *ACM Symposium on Principles of Programming Languages*, pages 121–132, January 1984.

[Joh85] T. Johnsson. Lambda lifting: transforming programs to recursive equations. In *Proceedings 1985 Conference on Functional Programming Languages and Computer Architecture*, Nancy, France, 1985.

[Mee84] L. Meertens. Algorithmics: Towards programming as a mathematical activity. In J. W. de Bakker, et. al., editors, *Mathematics and Computer Science*, North-Holland, 1984.

[Pey87] S. L. Peyton-Jones. *Implementing Functional Languages using Graph Reduction.* Prentice-Hall, 1987.

[SH82] M. R. Sleep and S. Holmström. A short note concerning lazy reduction rules of append. *Software Practice and Experience*, 12(11):1082–4, November 1982.

[Tho86] S. Thompson. Laws in miranda. In *ACM Symposium on Lisp and Functional Programming*, pages 1–12, August 1986.

[Tur81] D. A. Turner. Recursion equations as a programming language. In J. Darlington, P. Henderson, and D. Turner, editors, *Functional Programming and Its Applications*, Cambridge University Press, 1981.

[Tur85] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, Springer-Verlag, September 1985.

[Wad87] P. L. Wadler. Compiling pattern matching; List comprehensions. In [Pey87].