Transforming Sequences using Threaded Morphisms

A. N. Clark
Department of Computing, University of Bradford
Bradford, West Yorkshire, BD7 1DP
e-mail: a.n.clark@comp.brad.ac.uk, tel.: (0274) 385133

September 18, 1997

Abstract

Sequences are very useful structures in programming languages. Functional programming languages allow the convenient definition of transformations between sequence algebras and other types of data algebra. Sequence homomorphisms are an important class of such transformations. However, not all desired transformations can be expressed as homomorphisms. This paper describes a class of transformation which is a generalisation of homomorphisms: sequence morphisms with *threads*. This class is shown to contain a number of interesting sub-classes of transformation. The morphisms are described using a simple functional programming language and are applied to the design and implementation of a simple object-oriented programming language feature.

1 Introduction

This paper describes a class of sequence transformations or *morphisms*. The class is a generalisation of sequence homomorphisms and is shown to contain various interesting sub-classes. In order to describe this class of morphisms, we use a simple functional programming language. The aims of the paper are described in §1.1 and a brief overview of the characteristics of the functional programming language, used for exposition, is given in §1.2.

1.1 Overview

A sequence of values is either the empty sequence [] or is constructed from a value v and a sequence l using the right associative infix constructor $\_ : \_$ to produce $v : l$. A sequence of several values is represented as

$$v_1 : v_2 : \ldots : v_n : []$$

```
     :
     ^
    / \
  v1   :
       ^
      / \
    v2   ...
           \
            :
            ^
           / \
         vn   []
```

Figure 1: A typical sequence

as

$$[v_1, v_2, \ldots, v_n]$$

or, for the purposes of this paper, as a tree in figure 1. Given such a sequence we refer to the value v_i as the i^{th} sequence *element*.

The following sequence operators will be used: $\_ +\!\!+ \_$ which appends sequences, $\_ - \_$ which subtracts the elements of one sequence from another, *hd* which produces the head of a sequence, *tl* which produces the tail of a sequence, *map* which maps a function over a sequence, *zip* which maps a pair of sequences of the same length to a sequence of pairs or corresponding elements. Sequences of pairs are used as tables with the first component of each pair being the *key* and the second component being the corresponding value. Values are looked up in tables using the infix operator $\_ \bullet \_$ where $l \bullet k$ looks the key k up in the sequence of pairs l returning either the first associated value or ϵ. The infix operator $\_?\_$ is applied to the result of two lookup operations and returns the right hand operand only when the left is ϵ otherwise it returns the left hand operand.

Sequences of values are very useful in programming and are often used as the source of transformation to other data types. We seek general structure in these transformations so that they may be defined as general abstractions and talk of "mapping" a function over a list, of "folding" a list with respect to an operator, *etc.* The higher order operators which have been devised for sequence transformation are usually based upon the notion of *homomorphisms* between the sequence algebra and a target algebra. This is covered in §2 and is one type of general structure for sequence transformations which treats all the elements of the sequence independently.

It is not always possible to describe a required transformation as a collection of smaller independent transformations. Each component of such a transformation requires some context which it uses and modifies for use by the next transformation. For such transformations, there is an order in which the elements of the sequence must be transformed. We have termed these *threaded morphisms* and they are described in §3. §4 uses threaded morphisms to describe a non-trivial example, showing how classes are represented in a λ-calculus which has been enriched with environment reification. §5 describes related work and indicates future directions.

1.2 A Functional Programming Language

A functional programming language is used to describe the sequence morphisms in this paper. The language will not be given a precise semantics, but is intended to hold no surprises for the reader familiar with the class of programming languages including Haskell [8] and that described in [2]. Briefly, the following features are assumed to be part of the language, the reader is directed to [12] and [6] for more details.

An expression is either an identifier, a constant, an (infix) application, a λ-function, an **if**-expression, a **let**-expression, a **where**-expression, a **case**-expression, a tuple or a sequence. Both **let**- and **where**-expressions may define (curried) functions by introducing their arguments on the left hand side of the defining =. Programmer defined names are introduced in λ-functions, **case**-expressions, **let**-expressions and **where**-expressions. Wherever a programmer defined name is allowed, a pattern may be used. A pattern is either a constant, a name, a constructor applied to a tuple of patterns or simply a tuple of patterns. A pattern is used to test a value and to deconstruct a value into its components. The name _ is used as a pattern where we are not interested in the corresponding value. Top level definitions are introduced using a **let**-expression without a body; such definitions are recursive. Functions may be *overloaded* by giving extra definitions for the same name. Infix operators may be partially applied, as in $(+1)$.

The following operators are used: tuple accessors *1st*, *2nd* and *3rd*; the identity combinator **I**, the constant combinator **K** and the paradoxical combinator **Y**; $\_ \circ \_$ is infix function composition.

We informally use $e_1 \longrightarrow e_2$ to mean: if we were to inspect a sequence of snapshots of the execution regime for the functional programming language applied to expression e_1 we might expect to encounter the (usually simpler) expression e_2.

2 Sequence Homomorphisms

Suppose that we have a data type T whose values are freely constructed as either ϵ or a value x and a value t of type T composed using the infix constructor $\_ \otimes \_$, then a typical value of type T will be

$$x_1 \otimes x_2 \otimes \ldots \otimes x_n \otimes \epsilon$$

```
     :             f1(:)
     ^               ^
    / \             / \
 v1   :   -->  f2(v1) f1(:)
      ^                 ^
     / \               / \
   v2 []           f2(v2) f3([])
```

Figure 2: A typical homomorphism

Furthermore, suppose that given a typical sequence, as described in §1, for any sequence element v_i we can produce a value x_i, then a systematic rule for transforming sequences to T values is: replace all occurrences of : by $\otimes$; replace the single occurrence of [] by ϵ; transform each v_i to a corresponding x_i. Such a general transformation is called a *homomorphism*.

Sequence homomorphisms can be represented as triples (f_1, f_2, f_3) where f_1 transforms the : constructor to a binary operator $\otimes$, f_2 transforms each sequence element v_i to a value x_i and f_3 transforms the empty sequence [] to an "empty" value ϵ. Given such a triple we can invent an application rule for it

$$
\begin{array}{l}
\textbf{let}\,(f_1, f_2, f_3)(s) = \\
\quad\quad \textbf{case}\, s\, \textbf{of} \\
\quad\quad\quad\quad [] \Rightarrow f_3(s) \\
\quad\quad\quad\quad x : s' \Rightarrow f_1(:)(f_2(x), (f_1, f_2, f_3)(s')) \\
\quad\quad \textbf{end}
\end{array}
$$

Figure 2 shows how a typical homomorphism transforms a two element sequence. Homomorphisms compose using the infix operator $\_ \circ \_$ defined below

$$\textbf{let}\ (f_1, f_2, f_3) \circ (g_1, g_2, g_3) = (f_1 \circ g_1, f_2 \circ g_2, f_3 \circ g_3)$$

The identity transformation m_1 is an example of a homomorphism

$$\textbf{let}\ m_1 = (\mathbf{I}, \mathbf{I}, \mathbf{I})$$

There are many different useful sequence homomorphisms, such as m_2 which adds up all elements of a numeric sequence

$$\textbf{let}\ m_2 = (\mathbf{K}(+), \mathbf{I}, \mathbf{K}(0))$$

m_3 which adds up all the lengths of a sequence of sequences

$$\textbf{let}\ m_3 = (\mathbf{K}(+), \#, \mathbf{K}(0))$$

and m_4 which reverses a sequence

$$\textbf{let } m_4 = (\mathbf{K}(\lambda(x,s).s \mathbin{+\!\!+} [x]), \mathbf{I}, \mathbf{I})$$

In general, a homomorphism will "flatten out" some of the structure which is present in the sequence and it will not be possible to determine the original value from the result of transforming it. An example of this is seen when we transform a sequence of sequences using the homomorphism m_3

$$\textbf{let } x_1 = [[1,2,3],[4,5,6,7],[8,9,10]]$$

$$\begin{aligned} m_3(x_1) &\longrightarrow \#([1,2,3]) + \#([4,5,6,7]) + \#([8,9,10]) + 0 \\ &\longrightarrow 3+4+3+0 \\ &\longrightarrow 10 \end{aligned}$$

The result of the transformation, 10, is the sum of the component sequence lengths but it does not reflect any structure as to how the final result arose. Suppose that after transforming with m_3 we want to determine the maximum length which contributed to the value 10 using the homomorphism m_4

$$\textbf{let } m_4 = (\mathbf{K}(max), \mathbf{I}, \mathbf{I})$$

Now if we use m_3 to transform a sequence, m_4 can not be used since the structure of the original sequence is lost. On the other hand, if we use the composition $m_4 \circ m_3$ which produces the maximum length 4, we have lost the sum of the lengths 10.

In order to preserve the *inherited* structure from a sequence s, which is transformed $m(s)$, we define an operator, F_1, to be applied to a sequence, producing a new value $F_1(s)$. The new value is essentially the same as s except it may be transformed using homomorphisms m and the structure is preserved. In order to use a homomorphism m to transform a new value, it must also be transformed using F_1 to produce a new mapping $F_1(m)$. The operator F_1 has the properties of a *functor* [1].

$$\begin{aligned} &\textbf{let } F_1(s)(f_1,f_2,f_3) = \\ &\quad \textbf{case } s \textbf{ of} \\ &\quad\quad [\,] \Rightarrow f_3(s) \\ &\quad\quad x : s' \Rightarrow f_1(:)(f_2(x), F_1(s')(f_1,f_2,f_3)) \\ &\quad \textbf{end} \end{aligned}$$

$$\textbf{let } F_1(m_1)(s)(m_2) = s(m_2 \circ m_1)$$

The definition of F_1 has two parts. The first part defines what happens when F_1 is applied to a sequence s. The result, $F_1(s)$, is referred to as a *lifted* sequence and is an operator which is applied to a morphism, m, and returns the transformed sequence $m(s)$. The second part defines what happens when F_1 is applied to a sequence homomorphism, m_1. The result, $F_1(m_1)$, is referred to as a *lifted* morphism and is an operator which is applied to a lifted sequence s and returns a lifted sequence (an operator whose argument is a sequence homomorphism m_2).

Let $F_1(\alpha)$ represent the type of the values produced by applying the first definition of F_1 above, to sequences of type α. Let $\alpha \to \beta$ represent the type of homomorphisms from sequences of type α to some type β. The type of the result of applying the second definition of F_1 above to a homomorphism of type $\alpha \to \beta$ is $F_1(\alpha) \to F_1(\beta)$.

Now we can produce a lifted sequence

$$\textbf{let } x_2 = F_1(x_1)$$

the original sequence is recovered by applying the lifted sequence to the identity morphism, $x_2(m_1) \longrightarrow x_1$. The new value is produced by transforming x_2 with a lifted homomorphism

$$\textbf{let } x_3 = F_1(m_2)(x_2)$$

The sum of the lengths is produced by applying x_3 to the identity morphism $x_3(m_1) \longrightarrow 10$. The maximum sum of the lengths is produced by either applying x_3 to the homomorphism m_3, $x_3(m_3) \longrightarrow 4$ or by transforming x_3 using a lifted homomorphism and then applying this to the identity morphism $F_1(m_3)(x_3)(m_1) \longrightarrow 4$.

3 Threaded Sequence Morphisms

Not all transformations which we wish to perform on sequences can be described by a collection of independent transformations on the elements of the sequences. More complex transformations require contextual information where the context depends upon the position of an element of the sequence. Examples of this type of transformation are: merging one sequence into another and producing a sequence of the sums of prefixes or suffixes of a numeric sequence. In the first case, it is necessary to know the position in the sequence in order to pair up elements at the same positions. In the second case it is necessary to know the position in the sequence in order to add up all of the previous or succeeding elements.

We propose a generalisation of homomorphisms described in §2 which are morphisms with *threads*. A thread is a value which is passed through a morphism as a sequence is transformed and may be affected be individual components of the transformation. §3.1 describes morphisms which have downward threads, *i.e.* values which are passed through the transformation of successive elements from the head of the sequence to the tail. §3.2 describes morphisms which have upward threads, *i.e.* values which are passed through the transformation of successive elements from the tail of the sequence to the head. Finally, §3.3 describes morphisms which have both downward and upward threads.

```
  :            v  f1(:)
  ^             \   ^
 / \             \ / \
v1  :  -->  f2(v1,\)  f1(:)
    ^              \    ^         _
   / \              \  / \        /|
  v2 []        f2(v2,\)   f3([],/)
                      \        /
                       \______/
```

Figure 3: A typical morphism with a downward thread

3.1 Downward Threads

A sequence morphism with a downward thread is a triple (f_1, f_2, f_3) with the following application rule

$$
\begin{array}{l}
\textbf{let } (f_1, f_2, f_3)(s, v) = \\
\quad \textbf{case } s \textbf{ of} \\
\quad\quad [] \Rightarrow f_3(s, v) \\
\quad\quad x : s' \Rightarrow \ \textbf{let } (x', v') = f_2(x, v) \\
\quad\quad\quad\quad\quad\ \ \textbf{in } f_1(:)(x', (f_1, f_2, f_3)(s', v')) \\
\quad \textbf{end}
\end{array}
$$

Compared with the application rule for sequence homomorphisms in §2, the rule for downward threads has an extra parameter v which represents the thread. Each component x of the sequence is transformed before the rest of the sequence s' and the value of the thread is supplied to f_2 producing a new thread value v' which is supplied to the transformation of the rest of the sequence s'. Figure 3 shows a typical morphism with a downward thread applied to a two element sequence.

The following morphism

$$\textbf{let } m_5 = (\mathbf{I}, \lambda(x_1, x_2 : l).((x_1, x_2), l), \lambda([], []).[])$$

will merge a sequence into another sequence, for example

$$\textbf{let } x_4 = [1, 2, 3]$$

$$\textbf{let } x_5 = [4, 5, 6]$$

$$m_5(x_4, x_5) \longrightarrow [(1, 4), (2, 5), (3, 6)]$$

The following morphism

$$\textbf{let } m_6 = (\mathbf{I}, \lambda(x,v).(v, x+v), \lambda([], v).v : [])$$

will transform a sequence of numbers to a sequence of prefix sums and also add the complete sum onto the end of the sequence, for example

$$m_6(x_4, 0) \longrightarrow [0, 1, 3, 6]$$

$$m_6(x_5, 0) \longrightarrow [0, 4, 9, 15]$$

Now, suppose that we define a sequence morphism with a downward thread, which produces the sum of the prefix sums,

$$\textbf{let } m_7 = (\mathbf{K}(+), \lambda(x,v).(v, x+v), \lambda([], v), v)$$

Applying this morphism to a sequence will lose the structure of the sequence, as argued in §2. Furthermore, when morphisms are composed it is not possible to separate different threads for each layer of morphism. We will deal with these two problems by defining a functor F_2 which lifts both sequences and morphisms with downward *fibers*. A fiber is made up of multiple threads and is rotated between the morphisms. A morphism with a single thread is transformed to one with a rotating fiber using the operator *fiber*

$$\begin{array}{l}
\textbf{let } \mathit{fiber}(f_1, f_2, f_3) = (f_1, f_2', f_3') \\
\quad \textbf{where} \\
\quad\quad f_2'(x, v : l) = \textbf{let } (x', v') = f_2(x, v) \textbf{ in } (x', l +\!\!+ [v']) \\
\quad\quad f_3'(x, v : l) = \textbf{let } (x', v') = f_3(x, v) \textbf{ in } (x', l +\!\!+ [v'])
\end{array}$$

Now the functor F_2 is defined below

$$\begin{array}{l}
\textbf{let } F_2(s)((f_1, f_2, f_3), l) = \\
\quad\quad \textbf{case } s \textbf{ of} \\
\quad\quad\quad\quad [] \Rightarrow f_3(s, l) \\
\quad\quad\quad\quad x : s' \Rightarrow \ \textbf{let } (x', l') = f_2(x, l) \\
\quad\quad\quad\quad\quad\quad\quad\quad \textbf{in } f_1(:)(x', F_2(s')((f_1, f_2, f_3), l')) \\
\quad\quad \textbf{end}
\end{array}$$

$$\textbf{let } F_2(v)(m_1)(s)(m_2, l) = s(m_2 \circ m_1, v : l)$$

The functor F_2 has two parts to its definition. The first part defines what happens when F_2 is applied to a sequence s. The result, a lifted sequence, is an operator which expects a pair (m, l) where m is a morphism with a downward rotating fiber and l is the fiber. The second part defines what happens when F_2 is applied to a value v and a morphism m with a downward rotating fiber. The result, a lifted morphism, is an operator which is applied to a lifted sequence s and returns a lifted sequence.

F_2 is used to compose m_7 and m_5. Firstly, the sequence is lifted

$$\textbf{let } x_6 = F_2(x_5)$$

The original sequence is returned by supplying it with the identity morphism which has been transformed for fibers

$$x_6(\mathit{fiber}(m_1)) \longrightarrow x_5$$

Then, x_6 is transformed using m_7 to the sum of all the prefixes

$$\textbf{let } x_7 = F_2(0)(m_7)(x_6)$$

The sum is returned by supplying x_7 with the identity morphism

$$x_7(\mathit{fiber}(m_1)) \longrightarrow 10$$

Finally, x_7 is transformed using m_5 and merged with x_5

$$\textbf{let } x_8 = F_2(x_5)(m_5)(x_7)$$

and the merged value released by supplying the identity morphism

$$x_8(\mathit{fiber}(m_1)) \longrightarrow [(0,4),(1,5),(3,6)]$$

This is just one variation on the theme of morphisms with downward threads, other variations include: having a single thread which is shared between all composed morphisms; having the f_3 component of the morphism return an output value for the thread which is passed back as the *seed* value for successive morphisms (see §3.2); having indexable threads where a single morphism may affect either no threads, a single thread or multiple threads as desired.

Sequence morphisms with downward threads can be viewed as a generalisation of sequence homomorphisms and the functor F_2 can be made to implement F_1. Given a sequence homomorphism m, a morphism with a downward thread is produced using the operator *down*

$$\begin{array}{l} \textbf{let } \mathit{down}(f_1, f_2, f_3) = (f_1, f_2', f_3') \\ \quad \textbf{where} \\ \quad f_2'(x, v) = (f_2(x), v) \\ \quad f_3'(x, v) = (f_3(x), v) \end{array}$$

Now m can be applied to a sequence using F_2

$$\mathit{1st}(F_2(s)(\mathit{down}(m), [])) = F_1(s)(m)$$

and similarly for the morphism component of F_2, where any value may be supplied as the first argument v.

```
                    _
  :            |\  f1(:)
  ^              \    ^
 / \              \  / \
v1  :  -->  f2(v1,\)  f1(:)
    ^                \    ^          v
   / \                \  / \        /
  v2 []          f2(v2,\)   f3([],/)
                       \        /
                        \______/
```

Figure 4: A typical morphism with an upward thread

3.2 Upward Threads

A sequence morphism with an upward thread is a triple (f_1, f_2, f_3) with the following application rule

$$
\begin{array}{l}
\textbf{let } (f_1, f_2, f_3)(s, v) = \\
\quad \textbf{case } s \textbf{ of} \\
\quad\quad [] \Rightarrow f_3(s, v) \\
\quad\quad x : s' \Rightarrow \textbf{ let } (s'', v') = (f_1, f_2, f_3)(s', v) \textbf{ in} \\
\quad\quad\quad\quad\quad\;\; \textbf{let } (x', v'') = f_2(x, v') \\
\quad\quad\quad\quad\quad\;\; \textbf{in } (f_1(:)(x', s''), v'') \\
\quad \textbf{end}
\end{array}
$$

The application rule is similar to that which is defined in §3.1 except that the thread value v is passed to the end of the sequence where it is modified by f_3 and returned. This way the thread value taked part in the transformation of each element of the sequence from the tail to the head. Figure 4 shows a typical morphism with an upward thread applied to a two element sequence.

The morphism m_8, defined below, is an example of a morphism with an upward thread. The result of applying m_8 to a sequence of numbers is the sequence in which each element has been replaced by its suffix sum

$$\textbf{let } m_8 = (\mathbf{I}, \lambda(x, v).(v, x + v), \mathbf{I})$$

note that these morphisms return a pair which is the transformed sequence and the final value of the thread, *e.g.*

$$m_8(x_4, 0) \longrightarrow ([5, 3, 0], 6)$$

The morphism m_9 will merge the reverse of a sequence with another

$$\textbf{let } m_9 = (\mathbf{I}, \lambda(x_1, x_2 : l).((x_1, x_2), l), \mathbf{I})$$

for example

$$m_9(x_4, x_5) \longrightarrow ([(1,6),(2,5),(3,4)], [])$$

As is argued in §2 and §3.1, the application rule for a morphism with an upward thread loses the structure of the original sequence and prevents further useful transformations from being applied. We define a functor, F_3, which gets around this problem by retaining the structure in the transformations.

Before defining F_3 we note that there are a number of variations which arise when deciding how to deal with threads which are transformed by composed morphisms. §3.1 describes some variations and defines a functor F_2 which uses rotating fibers. This section defines a functor F_3 which *re-threads* the final value of a thread from one morphism as the seed value for the next morphism. An example application is the morphism which takes a sequence of numbers, adds them up and then replaces each element in the sequence with the sum. This is achieved using two morphisms with upward threads. The first morphism acts as identity on the sequence and produces the sum of the elements as the final value of the thread. The second morphism is seeded with the sum and replaces all the values in the sequence with the value of the thread.

The functor F_3 is defined as follows

$$
\begin{array}{l}
\textbf{let } F_3(s)((f_1, f_2, f_3), l) = \\
\quad \textbf{case } s \textbf{ of} \\
\quad\quad [] \Rightarrow \textbf{let } (o, \_, []) = f_3([], s, l) \textbf{ in } o \\
\quad\quad x : s' \Rightarrow \begin{array}[t]{l}
\textbf{let } l_1 = F_3(s)((f_1, f_2, f_3), l) \textbf{ in} \\
\textbf{let } (l_2, \_, []) = f_2([], x, \mathit{map}(\mathit{tl})(l_1)) \textbf{ in} \\
\textbf{let } \begin{array}[t]{l} l_3 = \mathit{map}(\mathit{1st})(l_1) \\ l_4 = \mathit{map}(\mathit{1st})(l_2) \\ l_5 = \mathit{map}(\mathit{2nd})(l_2) \end{array} \\
\textbf{in } \mathit{zip}(\mathit{map}((f_1)(:))(\mathit{zip}(l_3, l_4)), l_5)
\end{array} \\
\quad \textbf{end}
\end{array}
$$

$$\textbf{let } F_3(m_1)(s)(m_2, l) = s(m_2 \circ m_1, l)$$

The functor F_3 has two parts to its definition. The first part defines what happens when F_3 is applied to a sequence s. The result, a lifted sequence, is an operator which expects a pair (m, l) containing a morphism with upward threads and a list of values l. Consider the case where the supplied morphism is a composition $m_c \circ m_b \circ m_a$ and the list of values is $[v_0, v_1, v_2, v_3]$. The result of the application is a sequence of pairs

$$[(s_1, v_1), (s_2, v_2), (s_3, v_3)]$$

The transformation of s proceeds as follows. Firstly, the sequence s is transformed using the morphism m_a with an initial value v_0 for the thread. This produces a pair (s_1, v_1) which is the transformed sequence s_1 and the final value of the thread v_1. Although the value s_1 is not necessarily a sequence, it has inherited the structure of the sequence s and may be transformed using sequence morphisms. The next step is to transform the value s_1 (with its inherited structure) using the morphism m_b using the final value of the thread from the previous transformation, v_1, as the initial value for the new thread. This produces a pair (s_2, v_2) which is the transformed sequence, s_2, and the final value of the thread, v_2. Again, the value s_2 is not necessarily a sequence, but it will inherit the structure of s and therefore can be transformed using sequence morphisms. Finally, the value s_2 is transformed using the sequence morphism m_c and the final value of the thread produced by the previous transformation, v_2, as the initial value for the new thread. This produces a pair (s_3, v_3) which is the final result of transforming s with the transformation $m_c \circ m_b \circ m_a$ using v_0 as the initial value of the thread.

The second part of the definition F_3 is applied to sequence morphisms with upward threads and produces a morphism for lifted sequences.

In order to use the functor F_3, the morphisms must have a particular structure. Each morphism is a triple (f_1, f_2, f_3) as before, but the f_2 and f_3 components must be transition functions for state transition machines which are defined as follows. Each machine has states which are of the form

$$(o, v, i)$$

where: o is a sequence of pairs (v_1, v_2) consisting of a transformed sequence element v_1 and a transformed thread value v_2, each pair in the sequence is produced by successive morphisms; v is a sequence element, to be transformed by the next morphism; i is a list of thread values to be consumed by successive morphisms. A transition performed by a machine has the following format

$$(o, v_1, v_2 :: i) \longmapsto (o \mathbin{+\!\!+} [(v_1', v_2')], v_1', i)$$

where (v_1', v_2') is the pair of values produced by one step in the morphism using the pair (v_1, v_2). The value v_1' becomes the value which is to be transformed by the next morphism.

The list of seed values l which is supplied to the operator F_3 is found using a fixed point. The list is initially supplied as $v_0 : \perp$ where $\perp$ represents the rest of the seed values which have yet to be calculated. Each transformation which uses this list of initial values, produces a list where the initial element of the list depends only on v_0 and successive elements of the list each depend upon their predecessor. Thus, the resulting list of pairs are produced as $(s_1, v_1) : \perp$, whereupon the next component of the input list is known $v_0 : v_1 : \perp$. By applying this argument to successive pairs of the input and output we argue that the result may be constructed using a fixed point.

Given a sequence s, a morphism with a downward thread m and an initial value v, the transformed sequence is produced using the operator *tie*

$$\begin{array}{l}
\textbf{let } \mathit{tie}(s),(m,v) = \\
\quad \textbf{let } _ +\!\!+ [(s',v')] = \mathbf{Y}(\lambda l.s(m,v:(\mathit{map}(\mathit{2nd})(l)))) \\
\quad \textbf{in } (s',v')
\end{array}$$

Now we wish to define morphisms which add up all the values in a list and adds this value to each element.

$$\begin{array}{l}
\textbf{let } m_{10} = (\mathbf{I},g,h) \\
\quad \textbf{where} \\
\qquad g(o,v_1,v_2:l) = (o +\!\!+ [(v_1,v_1+v_2)],v_1,l) \\
\qquad h(o,[],v:l) = (o +\!\!+ [([],v)],[],l))
\end{array}$$

$$\begin{array}{l}
\textbf{let } m_{11} = (\mathbf{I},g,h) \\
\quad \textbf{where} \\
\qquad g(o,v_1,v_2:l) = (o +\!\!+ [(v_1+v_2,v_2)],v_1,l) \\
\qquad h(o,[],v:l) = (o +\!\!+ [([],v)],[],l))
\end{array}$$

The sequence x_4 is lifted using F_3

$$\textbf{let } x_9 = F_3(x_4)$$

transformed with the morphism m_{10}

$$\textbf{let } x_{10} = F_3(m_{10})(x_9)$$

the sum of the elements in the sequence x_4 is produced by supplying x_{10} with the identity morphism

$$\mathit{2nd}(\mathit{tie}(x_{10})(m_1,0)) \longrightarrow 6$$

x_{10} may be transformed using m_{11} which will add the sum of the elements on to each element

$$\textbf{let } x_{11} = F_2(m_{11})(x_{10})$$

and the final result is produced by supplying the identity morphism

$$\mathit{1st}(\mathit{tie}(x_{11})(m_1,0)) \longrightarrow [7,8,9]$$

Another example of the use of this morphism is to replace all the elements of a sequence with the maximum element. This is achieved as follows

$$\begin{array}{l}
\textbf{let } m_{12} = (\mathbf{I}, g, h) \\
\quad \textbf{where} \\
\qquad g(o, v_1, v_2 : l) = (o +\!\!+ [(v_1, max(v_1, v_2))], v_1, l) \\
\qquad h(o, v_1, v_2 : l) = (o +\!\!+ [(v_1, v_2)], v_1, l)
\end{array}$$

$$\begin{array}{l}
\textbf{let } m_{13} = (\mathbf{I}, g, h) \\
\quad \textbf{where} \\
\qquad g(o, _, v : l) = (o +\!\!+ [(v, v)], v, l) \\
\qquad h(o, v_1, v_2 : l) = (o +\!\!+ [(v_1, v_2)], v_1, l)
\end{array}$$

$$\mathit{1st}(\mathit{tie}(F_3(m_{12})(F_3([1,2,3,44,5,6,7])))(m_{13}, 0))$$

$$\longrightarrow [44, 44, 44, 44, 44]$$

Sequence morphisms with upward threads can be viewed as generalisations of sequence homomorphisms and the functor F_3 can be made to implement the functor F_1. Given a sequence homomorphism m, a morphism with an upward thread is produced using the operator *up*

$$\begin{array}{l}
\textbf{let } \mathit{up}(f_1, f_2, f_3) = (f_1, f_2', f_3') \\
\quad \textbf{where} \\
\qquad f_2'(o, v, i) = (o +\!\!+ [(f_2(v), v)], f_2(v), i) \\
\qquad f_3'(o, v, i) = (o +\!\!+ [(f_3(v), v)], f_3, (v), i)
\end{array}$$

Now m can be applied to a sequence using F_3

$$\mathit{1st}(\mathit{tie}(F_3(s))(\mathit{up}(m), \perp)) = F_1(s)(m)$$

3.3 Complete Threadings

§3.1 describes sequence morphisms with doward threads and §3.2 describes sequence morphisms with upward threads. Both of these types of morphism have been shown to be generalisations of sequence homomorphisms. This section merges the notions of upward and downward threaded morphisms into a single bi-directional or *completely* threaded sequence morphism.

In §3.1 and §3.2 it has been noted that there are different variations on the basic idea of a threaded morphism, for example fibers and successively threaded morphisms. This section describes a simple type of completely threaded sequence morphism which is used in §4.

The definition of the functor F for completely threaded sequence morphisms

```
  :            v1 _ f1(:)
  ^             \|\   ^
 / \             \ \ / \
v1  :  -->  f2(v1,\,\)  f1(:)
    ^              \ \    ^
                                     _
  / \               \ \  / \      v2 /|
  v2 []        f2(v2,\,\)   f3([],/,/)
                      \ \        / /
                       \ \______/ /
                        \________/
```

Figure 5: A typical morphism with bi-directional threads

is defined below

$$
\begin{array}{l}
\textbf{let } F(s)((f_1, f_2, f_3), v_1, v_2) = \\
\quad \textbf{case } s \textbf{ of} \\
\quad\quad [] \Rightarrow f_3(s, v_1, v_2) \\
\quad\quad x : s' \Rightarrow \\
\quad\quad\quad \textbf{let rec } (x', v_1', v_2'') = f_2(x, v_1, v_2') \\
\quad\quad\quad\quad\quad\quad\;\; (s'', v_1'', v_2') = F(s')((f_1, f_2, f_3), v_1', v_2) \\
\quad\quad\quad \textbf{in } (f_1(:)(x', s''), v_1'', v_2'') \\
\quad \textbf{end}
\end{array}
$$

$$
\textbf{let } F(m_1)(s)(m_2, v_1, v_2) = s(m_2 \circ m_1, v_1, v_2)
$$

There are two parts to the definition of the functor F. The first part describes what happens when F is applied to a sequence s. The result, a lifted sequence, is an operator which is applied to a completely threaded morphism and a pair of values, v_1 and v_2, which are the values of the downward and upward threads respectively. F does not implement any fancy thread mechanisms, such as rotating fibers or successive threading. The downward thread value is supplied to each successive element transformation in the sequence from the head to the tail and the upward thread value is supplied to each successive element transformation in the sequence from the tail to the head.

The second part of the definition of F describes what happens when F is applied to a completely threaded morphism. Figure 5 shows the result of applying a typical completely threaded morphism to a two element sequence.

Completely threaded morphisms may be used to pass information down a transformation as well as construct information which is passed back as a result of performing the transformation. The example which is given in §3.2 of replacing all the elements of a numeric sequence with the maximum element can be performed using a single completely threaded morphism as shown in the definition of *repmax* below

$$\begin{array}{l} \textbf{let } \mathit{repmax}(s) = \mathit{1st}(\mathbf{Y}(\lambda(\_,v,\_).F(s)((\mathbf{I},g,\mathbf{I}),0,v))) \\ \quad \textbf{where } g(v_1,v_2,v_3) = (v_3, \mathit{max}(v_1,v_2), v_3) \end{array}$$

Completely threaded sequence morphisms represent a generalisation of sequence homomorphisms, sequence morphisms with downward threads and sequence morphism with upward threads. The following operator definitions, *completehom*, *completedown* and *completeup*, show how the respective morphism types are transformed to become complete

$$\begin{array}{l} \textbf{let } \mathit{completehom}(f_1,f_2,f_3) = (f_1,f_2',f_3') \\ \quad \textbf{where} \\ \qquad f_2'(v_1,v_2,v_3) = (f_2(v_1),v_2,v_3) \\ \qquad f_3'(v_1,v_2,v_3) = (f_3(v_1),v_2,v_3) \end{array}$$

$$\begin{array}{l} \textbf{let } \mathit{completedown}(f_1,f_2,f_3) = (f_1,f_2',f_3') \\ \quad \textbf{where} \\ \qquad f_2'(v_1,v_2,v_3) = \textbf{let } (v_1',v_2') = f_2(v_1,v_2) \textbf{ in } (v_1',v_2',v_3) \\ \qquad f_3'(v_1,v_2,v_3) = \textbf{let } (v_1',v_2') = f_3(v_1,v_2) \textbf{ in } (v_1',v_2',v_3) \end{array}$$

$$\begin{array}{l} \textbf{let } \mathit{completeup}(f_1,f_2,f_3) = (f_1,f_2',f_3') \\ \quad \textbf{where} \\ \qquad f_2'(v_1,v_2,v_3) = \textbf{let } (v_1',v_3') = f_2(v_1,v_3) \textbf{ in } (v_1',v_2,v_3') \\ \qquad f_3'(v_1,v_2,v_3) = \textbf{let } (v_1',v_3') = f_3(v_1,v_3) \textbf{ in } (v_1',v_2,v_3') \end{array}$$

In each case F must be supplied with extra values for the threads which are not used. Since these values take no part in the transformation, they may be anything – even $\perp$.

Suppose that the type of a sequence element, or a value which can inherit the structure of a sequence, is represented as α and that a completely threaded morphism from one type of sequence to another is represented as $\alpha_1 \rightarrow \alpha_2$. $F(\alpha)$ represents the type of values which are constructed by applying F to a sequence of type α and $F(\alpha_1) \rightarrow F(\alpha_2)$ represents the type of morphisms for values of type $F(\alpha_1)$ to values of type $F(\alpha_2)$. We are interested in the circumstances under which the following diagram commutes

$$\begin{tikzcd} F(\alpha_1) \arrow[r, "F(m_1)"] \arrow[d, "F(m_2)"'] & F(\alpha_2) \arrow[d, "F(m_3)"] \\ F(\alpha_3) \arrow[r, "F(m_4)"'] & F(\alpha_4) \end{tikzcd}$$

Theorem 3.1 *The above diagram commutes when*

$$m_3 \circ m_1 = m_4 \circ m_2$$

Proof 3.1

$$
\begin{aligned}
&F(m_3)(F(m_1)(F(s))) \\
&\quad = F(m_3)(\lambda(m, v_1, v_2).F(s)(m \circ m_1, v_1, v_2)) \\
&\quad = \lambda(m, v_1, v_2).(\lambda(m, v_1, v_2).F(s)(m \circ m_1, v_1, v_2))(m \circ m_3, v_1, v_2) \\
&\quad = \lambda(m, v_1, v_2).F(s)(m \circ m_3 \circ m_1, v_1, v_2) \\
&\quad = \lambda(m, v_1, v_2).F(s)(m \circ m_4 \circ m_2, v_1, v_2) \\
&\quad = \lambda(m, v_1, v_2).(\lambda(m, v_1, v_2).F(s)(m \circ m_2, v_1, v_2))(m \circ m_4, v_1, v_2) \\
&\quad = F(m_4)(\lambda(m, v_1, v_2).F(s)(m \circ m_2, v_1, v_2)) \\
&\quad = F(m_4)(F(m_2)(F(s)))
\end{aligned}
$$

4 An Example Using Threads

This section describes an application of the complete threads described in §3.3. In §4.1 we specify a simple object-oriented language feature involving classes, objects, method lookup and inheritance. The feature is typical of object-oriented programming languages such as Smalltalk [7] and C++ [5]. These features are implemented in §4.2 using a functional programming language. Both the specification and implementation can be expressed using completely threaded sequence morphisms. We are interested in showing that the implementation is consistent with the specification and in §4.3 we use the proof of consistency from §3.3 to achieve this. Finally, §4.4 describes a number of variations which are possible on the theme of object-oriented programming language features and threaded morphisms.

4.1 Specification

A program binding *environment* is a sequence of pairs (i, v) where i is a program identifier and v is a program value. An identifier is looked up in an environment using the operator $\_\bullet\_$ as described in §1.2. A *class* is a sequence of pairs of the form (l, e) where l is a list of *attribute* names declared by the class and e is an environment of methods. A *method* is a pair (i, b) where i is an identifier which is the method argument and b is a program expression which is the method body. The sequence [] is the empty class which all classes inherit from, the sequence $p : s$ is a class where s is a *superclass* and p is an extension which adds attribute names and methods to s.

An *object* is a sequence of pairs of the form (e_1, e_2) where e_1 is an environment of attribute names bound to attribute values and e_2 is an environment of methods. The process by which a class is transformed into an object is called *instantiation*, each attribute name declared by the class is supplied with an initial value during instantiation, otherwise the structure of the object (or instance) reflects that of the class.

A *message*, consisting of a method name and an argument, is sent to an object. The name is looked up in the object pairs from the head of the object to the tail. The first method whose name matches the message name is activated and the position of the method in the object sequence is important because it defines which attribute names are visible. Given an object

$$p_1 : p_2 : \ldots : p_i : \ldots p_n : []$$

if a method is selected and activated from the pair p_i then the attribute names and values which are available to the body of the method are the concatenation of all the attribute names and values in the object $p_i : \ldots : p_n : []$. The activation of a method requires a method argument, an environment binding attribute names and a method body. The result of method lookup is described as a triple (i, e, b) which is a method argument i, attribute environment e and method body b.

Instantation of a class is performed by morphism m_1

$$\begin{array}{l} \textbf{let } m_1 = \mathit{completedown}(\mathbf{I}, g, h) \\ \quad \textbf{where} \\ \qquad g((l, e), v) = ((\mathit{zip}(l, \mathit{hd}(v)), e), \mathit{tl}(v)) \\ \qquad h([], ()) = ([], ()) \end{array}$$

Given a class

$$\textbf{let } c = [([i_1], [(i_2, (i_3, b_1))]), ([i_4], [(i_5, (i_6, b_2))])]$$

it is instantiated using

$$F(c)(m_1, ([1], ([2], ())))$$

A method whose name is i is looked up using the morphism m_2

$$\begin{array}{l} \textbf{let } m_2(i) = \mathit{completeup}(\mathbf{K}(?), f, \mathbf{K}(\epsilon, [])) \\ \quad \textbf{where} \\ \qquad f((e_1, e_2), e_3) = \\ \qquad\quad \begin{cases} ((i, e_1 + e_3, b), e_1 + e_3) & \text{when } e_2 \bullet i = (i, b) \\ (\epsilon, e_1 + e_3) & \text{otherwise} \end{cases} \end{array}$$

Given an object

$$\textbf{let } o = [([(i_1, 1)], [(i_2, (i_3, b_1))]), ([(i_4, 2)], [(i_5, (i_6, b_2))])]$$

the method named i_2 is looked up using the morphism m_2

$$F(o)(m_2(i_2), [])$$

producing the triple

$$(i_3, [(i_1, 1), (i_4, 2)], b_1)$$

4.2 Implementation

§4.1 specifies a simple object-oriented language feature using sequence morphisms. This section describes how this feature can be implemented in a simple functional programming language using environment reification. We will use a standard functional programming language with the following extensions:

- Lists of identifier/value pairs are used as programming language binding environments.
- The operator R is applied to a function closure and produces its binding environment.
- The operator I is applied to a pair (e, f) containing a binding environment e and a function closure f. The result is a new function closure which is the same as f except that the binding environment has been replaced with e.

The primitive components necessary to provide environment reification are discussed at more length in [11], [9] and [3]. The infix operator $\_ \hookrightarrow \_$ extends the binding environment of a function closure

$$\textbf{let } e \hookrightarrow f = I(e +\!\!\!+ (R(f)), f)$$

The operator *class* constructs classes from a superclass c and an extension x which defines attribute names and methods

$$\begin{array}{l} \textbf{let } \mathit{class}(c, x)(v_1, v_2) = \\ \quad \textbf{let } (o_1, a_1) = c(v_1) \textbf{ in} \\ \quad \textbf{let } (o_2, a_2) = (a_1 \hookrightarrow x)(v_2) \\ \quad \textbf{in } (o_2 +\!\!\!+ o_1, a_2 +\!\!\!+ a_1) \end{array}$$

When a class is instantiated, by applying it to a collection of initial values, it returns a pair (o, a) which consists of a binding environment o of function closures and a binding environment a of attribute values. The operator $\hookrightarrow$ is used in *class* to cause the extension function x to inherit the attribute values a_1 created when the superclass c is instantiated.

A typical extension function is as follows

$$\textbf{let } x = \mathbf{Y}\lambda f.\lambda(i_1, i_2, \ldots i_n).(E, R(\lambda().()) - R(f))$$

where the identifiers $i_1 \ldots i_n$ are the attribute names, E is an expression which constructs a binding environment for the methods and the expression $R(\lambda().()) - R(f)$ constructs just that binding environment containing the attribute names.

All classes ultimately inherit from the null class

$$\textbf{let } \mathit{nullclass}() = ([], [])$$

which returns the empty method and attribute environments when it is instantiated.

When the functional programming language computes with the operators described above, there will be computational occurrences of classes and extensions. It is beyond the scope of this paper to fully describe what these computational occurrences would look like, but we assume the following operators which are described in terms of their arguments and return values. The operator *classclo* is applied to a computational occurrence of an extension, x, and a class, c

$$\mathit{classclo}(x, c)$$

and returns a new computational occurence of a class, as described above by the operator *class*. The operator *xclo* is applied to a list of attribute names, l, and an environment of lambda function expressions, e

$$\mathit{xclo}(l, e)$$

and produces a computational occurence of an extension as described by the operator x. The operator *buildclass* is defined as follows

$$\textbf{let } \mathit{buildclass}((l, e), c) = \mathit{classclo}(c, \mathit{xclo}(l, e))$$

Finally, the value *nullcassclo* is the computational occurrence of the *nullclass* operator defined above.

Given a computational occurrence of a class, constructed with *classclo*, the structure of the class will be complex, especially if the functional language has *desugared* all values, for example by uncurrying function definitions and expanding all patterns into pattern matching code. The instantiation of such a value requires explanation of the evaluation rules for function application, testing values against patterns, *etc.* However, if such a value is intended to be an implementation of a class described in §4.1, then we can use F to transform the original class and the structure will be retained. The following morphism transforms a class sequence to a computational occurrence

$$\textbf{let } m_3 = \mathit{completehom}(\mathbf{K}(\mathit{buildclass}), \mathbf{I}, \mathbf{K}(\mathit{nullclassclo}))$$

If the structure of the original class sequence is retained in the computational occurrence then instantiation (by application) is described using the following morphism

$$\begin{array}{l}
\textbf{let } m_4 = (\mathbf{K}(+\!\!+), f, \mathbf{K}([])) \\
\quad \textbf{where} \\
\quad\quad f((l_1, e_1), (l_2, v), e_2) = (\mathit{map}(g)(e_1), v, e_3) \\
\quad\quad\quad \textbf{where} \\
\quad\quad\quad\quad g(i_1, (i_2, b)) = (i_1, (i_2, e_3, b)) \\
\quad\quad\quad\quad\quad \textbf{where } e_3 = (\mathit{zip}(l_1, l_2)) +\!\!+ e_2
\end{array}$$

Finally lookup of a method named i is performed using the morphism m_5

$$\textbf{let } m_5(i) = \mathit{completehom}(\mathbf{K}(?), (\bullet i), \mathbf{K}(\epsilon))$$

4.3 Consistency

We wish to show that the implementation of the simple object-oriented feature is consistent with its specification. The theorem which is proved in §3.3 requires only that we prove a simpler theorem about the individual morphisms for this to be the case.

Theorem 4.1 *The implementation of classes, instantiation and method lookup is consistent with the specification, i.e.*

$$m_2 \circ m_1 = m_5 \circ m_4 \circ m_3$$

Proof 4.1

$$(1) \quad \mathit{1st}(m_2 \circ m_1) = \mathbf{K}(?) = \mathit{1st}(m_5 \circ m_4 \circ m_3)$$

$$(2) \quad \mathit{2nd}(m_2 \circ m_1)((l_1, e_1), (l_2, v), e_2)$$

$$\mathit{Let}\ e = (\mathit{zip}(l_1, l_2)) + e_2$$

$$\begin{aligned} &= \mathit{2nd}(m_2)((\mathit{zip}(l_1, l_2), e_1), v, e_2) \\ &= \begin{cases} ((i, e, b), v, e) & \mathit{when}\ e_1 \bullet i = (i, b) \\ (\epsilon, e) & \mathit{otherwise} \end{cases} \end{aligned}$$

$$\begin{aligned} &\mathit{2nd}(m_5 \circ m_4 \circ m_3)((l_1, e_1), (l_2, v), e_2) \\ &= \mathit{2nd}(m_5 \circ m_4)((l_1, e_1), (l_2, v), e_2) \\ &= \mathit{2nd}(m_5)(\mathit{map}(g)(e_1), v, e) \\ &\quad \textbf{where}\ g(i_1, (i_2, b)) = (i_1, (i_2, e, b)) \\ &= \begin{cases} ((i, e, b), v, e) & \mathit{when}\ e_1 \bullet i = (i, b) \\ (\epsilon, e) & \mathit{otherwise} \end{cases} \end{aligned}$$

$$(3) \quad \mathit{3rd}(m_2 \circ m_1) = \mathbf{K}(\epsilon, []) = \mathit{3rd}(m_5 \circ m_4 \circ m_3)$$

4.4 Variations

§4.1 specifies a simple object-oriented programming language feature, §4.2 implements the feature using a functional programming language with environment reification and §4.3 shows that the implementation is consistent with the specification.

The specification and implementation depends upon downward sequence morphisms for class instantation and upward sequence morphisms for attribute value inheritance. The language feature is very simple and does not require sophisticated threaded morphism techniques. There are a number of variations on this language feature which occur in currently available object-oriented programming languages. These include multiple inheritance where each class inherits from more than one parent, run-super where shadowed methods are made available by passing the tail portion of an object to an activated method, run-inner where the prefix portion of an object is passed to an activated method, self reference where the entire object is passed to an activated method. These mechanisms can be expressed using variations on the basic theme of threaded sequence morphisms, see [4] for more details of these mechanisms and for the application of threaded morphisms to the calculations performed by a programming language.

5 Conclusion, Related and Future Work

§2 describes sequence homomorphisms which are a useful sub-class of sequence transformations. §3 describes threaded morphisms which are a generalisation of sequence homomorphisms and which allow information to be threaded both down and up a sequence as it is being transformed. Such morphisms provide greater freedom to express transformations because they allow a sequence element to be transformed in context. It is useful to allow the transformed sequences to retain structure, providing scope for further sequence transformations to be applied. The structure is retained by defining a functor which lifts both sequences and threaded sequence morphisms. Threaded sequence morphisms have been used to specify and prove the consistency of an implementation of a simple object-oriented programming language feature.

This work is obviously related to monads [16] [15] where a monad is also viewed as a mechanism for plumbing thread-like values through program constructions. The work on monads has tended to focus on control issues such as state [15], input/output [13] and non-determinism [16], in which case the threads are used to hide away values which are usually thought of as being part of the language execution mechanism.

Both the work on monads and on the theory of lists [14] focus on the types of values and the properties which transformations must preserve. Although the work described in this paper focusses on the implementation of several different types of threaded morphism, there is some indication that a more rigorous treatment is possible from the declaration that the various F functions are all functors. This is an area for further work.

The completely threaded morphisms defined in §3.3 are similar to attribute grammars in the sense that both can be used to pass information down a given structure and construct information back up the same structure. Threaded sequence morphisms are more general that attribute grammars since they are not forced to pass information in any particular direction, see [10] for more details.

The work described in this paper limits itself to defining transformations on sequences. It is obvious that threaded morphisms can be defined for any algebraic data type; the use of sequences greatly simplified the explanations (and it is surprising what you can get away with just using sequences). The definition of threaded morphisms for more complex data types should be straightforward and it is likely that the extra structure provided by the data types will give rise to more sophisticated general thread mechanisms.

References

[1] Barr, M. & Wells, C. (1990) *Category Theory for Computing Science* Prentice Hall International Series in Computer Science.

[2] Bird, R. & Wadler, P. (1988) *Introduction to Functional Programming* Prentice Hall International Series in Computer Science.

[3] Clark, A. N. (1994) "A Layered Object-Oriented programming Language" *The GEC Journal of Research.* 11 **3** 173 – 180.

[4] Clark, A. N. (1995) *Semantic Primitives for OOPLs* PhD Thesis, Queen Mary and Westfield College, University of London.

[5] Ellis, M. A & Stroustrup B. (1990) *The Annotated C++ reference Manual* Addison Wesley.

[6] Field, A. J. & Harrison, P. G. (1988) *Functional Programming* Addison Wesley.

[7] Goldberg, A. & Robson, D. (1983) *Smalltalk-80 The Language and its Implementation* Addison-Wesley.

[8] Hudak, P. *et. al.* (1992) "Report on the Functional Programming Language Haskell, Version 1.2" *SIGPLAN Notices* 27.

[9] Jagannathan, S. (1994) "Metalevel Building Blocks for Modular Systems" *ACM TOPLAS* 16 **3** 456 – 492.

[10] Johnsson, T. (1987) "Attribute Grammars as a Functional Programming Paradigm." in *Conference on Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science no. 274, pp. 154 – 173, Springer-Verlag*

[11] Miller, J. S & Rozas, G. J. (1991) "Free Variables and First-Class Environments" *Lisp and Symbolic Computation: An International Journal* 4, 107 – 141.

[12] Peyton Jones, S. L. (1987) *The Implementation of Functional Programming Languages* Prentice Hall International Series on Computer Science.

[13] Peyton Jones, S. L. & Wadler, P. (1993) "Imperative functional programming" in *ACM Conference on Principles of Programming Languages, Charleston* 71 – 84.

[14] Spivey, M. "A Categorical Approach to the Theory of Lists" in *Lecture Notes in Computer Science no. 375, Springer-Verlag*

[15] Wadler, P. (1990) "Comprehending Monads" in *ACM Conference on Lisp and Functional Programming, Nice.*

[16] Wadler, P. (1992) "The Essence of Functional Programming" in *ACM Conference on the Principles of Programming Languages, 19*