MARIGOLD
A FUNCTIONAL, FLOW-GRAPH LANGUAGE

BY

RICHARD B. KIEBURTZ

OGC TR CS/E 83-005

Oregon Graduate Center (503) 645-1121
Dept. of Computer Science & Engineering
19600 N.W. Walker Road Beaverton, OR 97006

*Revised September, 1983.

Marigold – A functional, flow-graph language

Richard B. Kieburtz

Oregon Graduate Center
19600 NW Walker Road
Beaverton, OR 97006

Abstract[1]

Marigold is a new functional programming language that exploits the style of flow-graph specification. It has the power of a universal language, yet all programs appear as finite, reducible flow-graphs, without the use of recursion. Several examples of programs are given in the paper, and are contrasted with programs given in other styles.

The fundamental data type of Marigold is the stream. Stream objects are (in general) of unbounded extent, and arbitrarily high-order streams may occur. An evaluator must perform lazy evaluation of stream construction in order to obtain terminating computations.

A formal semantics, based upon the theory of partially-additive monoids, is outlined in the paper.

Key words and phrases: programming languages, functional programming, lazy evaluation, flow-graphs, formal semantics.

[1]The research reported here was supported by the OGC CS/E consortium with funding from Floating Point Systems and Tektronix.

1. Functions specified by flow-graphs

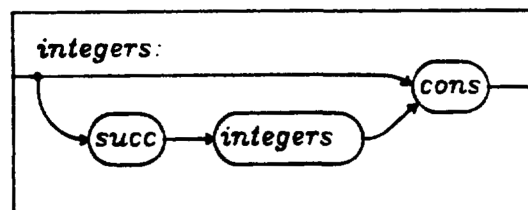
The idea of specifying programs by flow-graphs seems to be as old as computing. It is particularly attractive in a setting in which programs are intended to specify functions (i.e. there is no implicit, global state). No artificial distinction needs to be made between control and data flow -- the arcs of a flow-graph simply denote the flow of information from one operator to the next. Therefore it is surprising that there remain aspects of programming with flow-graphs that have not been fully explored -- in fact, have hardly been touched upon!

This is not to say that there have not been substantial and noteworthy contributions to the development of flow-graph languages. FGL [Kel80] is a full-blown functional flow-graph language that has been formally defined, implemented by an interpreter, and evaluated by its use in many examples. But the semantics of FGL is based upon its translation into the λ-calculus. It can achieve full expressiveness without the use of recursive definitions, but only by resort to higher-order functions and to the device of self-applied functions. While interesting from a theoretical point of view, we do not regard these devices as natural to use in a programming notation. Most programmers of flow-graph languages such as FGL make use of recursive definitions.

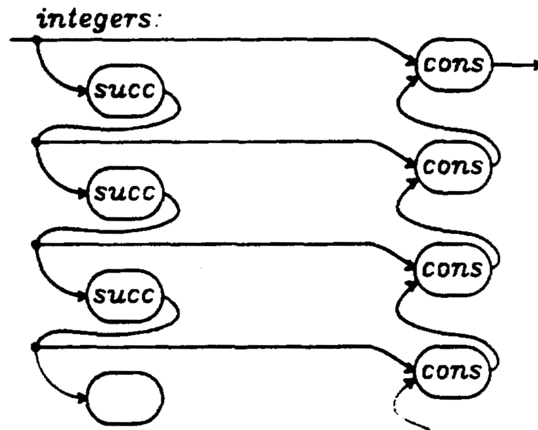
When recursive definitions are specified, the notation regresses to naming. A name is bound to a flow-graph, and the name is used to represent the graph in a recursive application. If the graph were to be fully instantiated by substituting for each name the graph it represents, then the occurrence of a recursive invocation would result in an infinite graph! The problem is illustrated by a familiar example. An infinite list of ascending integers generated from a single integer argument can be obtained by applying the recursively-defined function (given here in LISP notation)

(def *integers* (*lambda* (*n*)
    (*cons* *n* (*integers* (*succ* *n*)))))

Writing this definition as a flow-graph gives



in which the name of the graph, ***integers***, occurs on one of its nodes. Fully instantiating the graph leads to

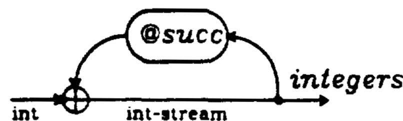


in which the diagram represents an infinite graph.

A solution to this problem is hinted at by Peter Henderson [Hen80]. He draws a finite graph by applying *succ* not to the argument *n* of *integers*, but to the entire list itself. Of course, *succ* is not quite the right function to apply to a list; it should be *mapcar 'succ'*. This gives the LISP definition

(def *integers* (*lambda* (*n*)
    (*cons n* (*mapcar 'succ'* (*integers n*)))))

and leads to a finite flow-graph,



where the symbol '@', read "apply-to-all", is equivalent to the LISP functional *mapcar*. The symbol ⊕ is the stream constructor, analogous to the list constructor *cons* of lisp. The stream constructor was introduced in FGL and named *fby*, which is pronounced as "followed by".

The list defined by applying *integers* to any finite integer argument is, of course, infinite. Such function definitions can only lead to terminating computations if list (or stream) construction is defined as a non-strict function, allowing lazy evaluation [HeM76, FrW76].

If the LISP definition is rewritten as a recursion equation,

integers n = (*cons n* (*mapcar 'succ'* (*integers n*)))

the form of this equation may strike the reader as unusual. We have become accustomed to writing recursion equations so that any applicative expression occurring in the right-hand side, even one that represents a recursive invocation, will be defined in a poset ordering before the applicative expression on the equation's

left side. In the equation above, the recursive invocation on the right is precisely the same applicative expression that appears on the left side. However, few readers would doubt for long that the above equation is well-founded.

The possibility of definitions such as the one above, with the attendant consequence of a finite flow-graph representation, motivates the question "are finite flow-graph representations sufficient to define all computable functions?" The flow-graph language Marigold is presented to lay this question to rest. As we shall see, it leads to a programming style that is in some cases quite different from those to which we have become accustomed. The style emphasizes iteration, not recursion. The paradigm used for recursive function definition is derived from primitive recursion schemes and the Kleene μ operator.

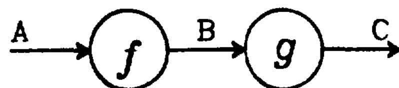
1.1. Flow-graph operators

The operators of a flow-graph language are the several kinds of nodes that it defines. There are just four kinds of operators in Marigold:

i) Function nodes:

represents a function $f : A \to B$.

Given functions $f : A \to B$, and $g : B \to C$, a pair of function nodes connected by an arc,

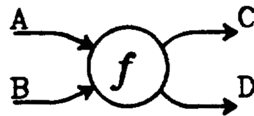


represents the composition of functions $f \circ g : A \to C$ (function composition is written here in diagrammatic order).

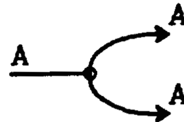
Multiple arcs originating or terminating on a common node represent multiple values, or equivalently, represent a value of a cartesian product type. Selection of components of a value of product type is implied by the routing of individual arcs. When there are multiple arcs incident upon a node, they may be annotated with integers to avoid ambiguity among the components of a product type. For example, a function of two arguments which also produces a result having two components of its value,

$f : A \times B \rightarrow C \times D$

could be represented by

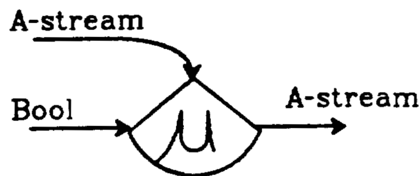


ii) Value-sharing nodes:

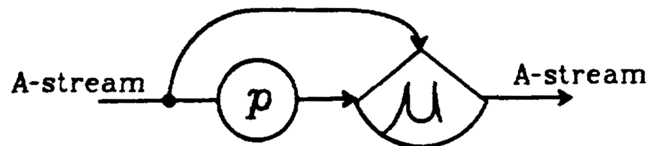


represents a common value appearing on multiple data-flow arcs.

iii) Recursive selection[2]:



represents a data-activated filter that is controlled by a stream of Boolean values at its control input. It transmits an element of its type *A−stream* input arc onto its output arc only when the element at the head of its control input stream is the Boolean value T. Recursive selection is used to form a graphical equivalent of Kleene's recursion operator, $\mu(p)$,

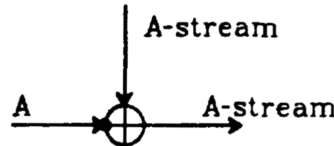


This graph represents a function that filters its input stream, admitting into its output stream only those elements which satisfy the control predicate p. As we shall see later, this scheme can be used to form a conditional selection

[2]Recursive selection is the only Marigold operator that does not have an exact counterpart in FGL. It can be simulated in FGL by a **while** loop.

(**if**... **then**... **else**...) scheme.

iv) Stream construction:

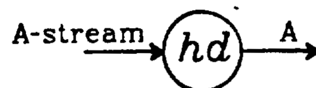


1.2. Streams are fundamental data types

In flow-graph notation, streams are the foundation for data types. An arc connecting two nodes of a flow-graph represents data, and can be labelled with the type of that data. Arcs may represent individual items, streams of items, streams of streams of items, etc. Thus the fundamental data types are n-order streams. The type of an individual item (such as a single Boolean value) is considered to be a 0-order stream.

Accordingly, Marigold has several polymorphically-typed operators on streams. In addition to stream construction, given above, these are:

Next element selection:



Rest of stream selection:

Apply-to-all:

$$f : A \rightarrow B$$

Boolean values are also representable in terms of streams. The null stream, denoted by <>, is the canonical representation for the Boolean value T. The Boolean value F has no canonical representation; any non-null stream is a representation for F. It is convenient to designate explicitly those streams used as Boolean values, so we define the predicate that tests for a null stream,

although it is equivalent to an identity function, or a vacant arc of a flow-graph

1.2.1. The expressive power of streams

There are several more or less obvious aspects of streams that must be kept in mind when a reader is interpreting Marigold program schemes. The value of a stream is an r.e. set; sometimes these sets are finite. Pairing an r.e. set with its index set gives the graph of a function; thus streams can be considered to represent partial recursive functions. A stream of streams can be considered to represent a function of (at least) two arguments, in curried form. For instance, if S is a second-order stream, we can say that it represents a function of two arguments, f_S. Selection of an element of S by index x yields a secondary stream, S_x, which represents $f_S\ x$, a function of a single argument. Further selection from S_x by index y yields an element $S_{x,y}$ which is the value produced by the (double) application $f_S\ x\ y$.

These observations have importance because Marigold is a language in which the only second-order functions are the operators defined above. The language does not provide functional abstraction, and functions are not first-class objects in Marigold. In a language based upon the λ-calculus this would severely restrict its expressive power. But Marigold programmers have streams to use in place of higher-order functions. When using streams, one tends to think in terms of r.e. sets and transformations upon sets, rather than of transformations upon functions, and to some programmers at least, this mode of thought seems more natural.

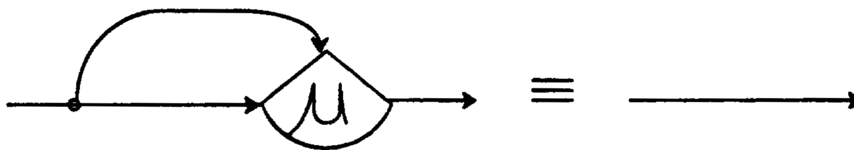
1.3. Well-formed flow-graphs

Not every graph that can be formed by interconnection of a set of nodes with directed arcs is to be considered a well-formed flow-graph. In fact, we impose a strong constraint upon the topology of a directed graph that will be given meaning as a program; namely that it must be *reducible*. There are six reduction rules that may be applied to test a graph for reducibility. In rules (a-b), the nodes may be of any of the four kinds listed above, subject only to the constraint that incident arcs have their heads and tails drawn consistently with the definition of each kind of node.

a)

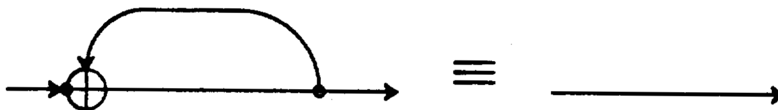
b)

c)



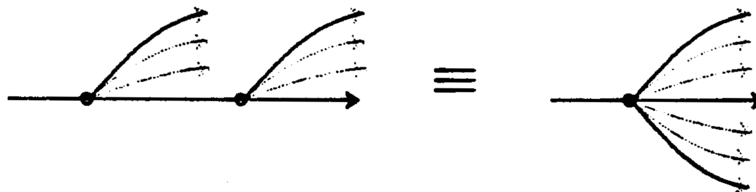
This rule is a special case of (a) + (b).

d)



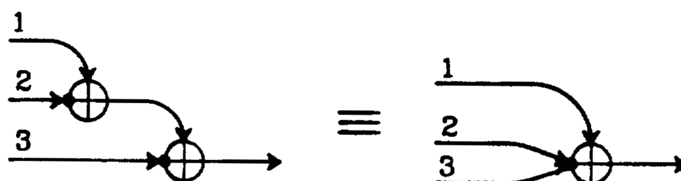
This rule applies only to a stream construction node.

e)



This rule applies only to value-sharing nodes.

f)



This rule applies only to stream-construction nodes.

A graph is said to be ***reducible*** if it can be reduced to a single arc by repeated application of rules (a-f). Reducible flow-graphs have been studied by Ullman and Hecht, and an alternate characterization is given in [Hec77].

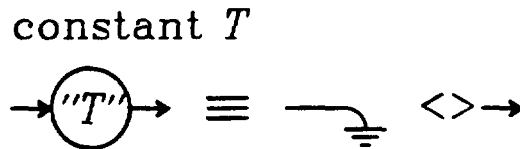
A flow-graph specification is well-formed if it is both reducible and well-typed. Both of these properties can be checked by static analysis of a flow-graph.

2. Programming in Marigold

To illustrate program construction with Marigold, let us first specify the elementary arithmetic functions, then give several more interesting examples. Our goal is to illustrate the power of the Marigold operators; thus we shall be very sparing in the use of primitive functions. We do, however, require a basis for the domain of objects that are represented on the arcs of graphs. A minimum basis is the null stream object.

2.1. Getting started

Marigold programs define functions which map n^{th}-order streams to m^{th}-order streams. In making such definitions it is often necessary to specify an initial value. Recall, however, that a flow-graph must be reducible to a single arc in order to constitute a well-formed Marigold graph. The reducibility requirement prohibits the appearance of values as nodes of a program flow-graph; we must instead use constant-valued functions, as is done in FP [Bac78]. Thus to use the Boolean value T explicitly in a program definition, we make a constant function,

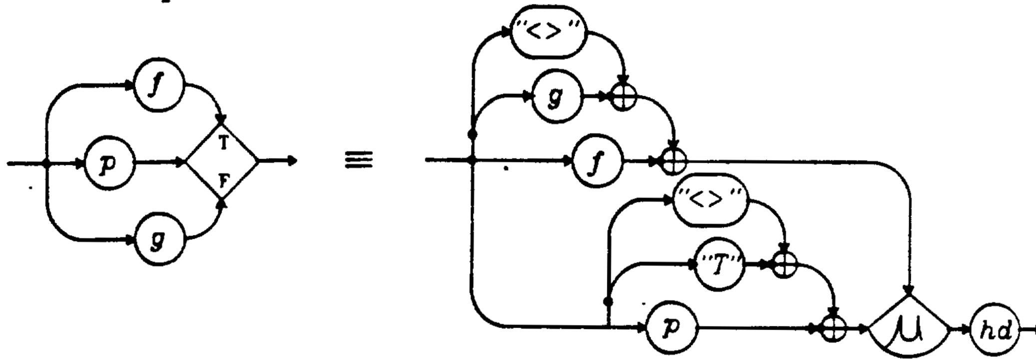


The generator for streams is the null stream. Often it is necessary to describe as well a singleton stream that contains one object. A stream containing the single object u is denoted by

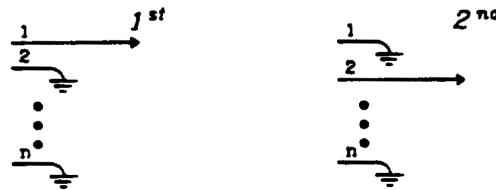
2.2. A convenient macro definition

Some programs use conditional selection among two (or more) alternative functions to apply to an input stream. Such a selection can be programmed, using the μ-operator, but the resulting graph is more complex to read than seems warranted. Therefore, in order to make the selection operation manifest, we shall define the following as equivalent graphs. The left side, which might be called

an **if-then-else** construction in a textual language, is not a new primitive of Marigold. Rather, the definition should be thought of as a macro-operator.



Many functions take multiple arguments which in Marigold, as in FP, are represented as single arguments of a cartesian product type. In a Marigold flow-graph, this is manifested by multiple arcs incident upon a function node. In order to express selection among the incident arcs, we shall use numbers as labels when convenient, and shall also introduce selector names (*a la* FP) as polymorphic functions defined by graph schemes such as:

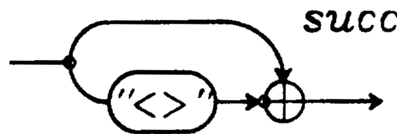


2.3. Generators for the natural numbers

To generate a representation for the natural numbers, there must be a zero element, for which we use the null stream object of Marigold,

zero : <>

and a successor function,



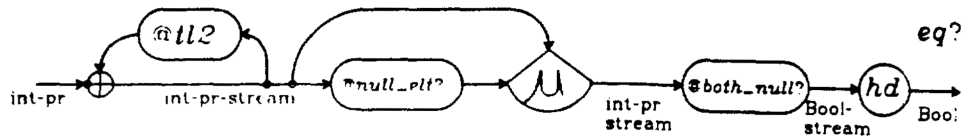
Non-zero numbers are generated by application of *succ* to preceding numbers, and are represented by finite sequences of the null stream:

0 : <> 1 : <<>> 2 : <<>,<>> 3 : <<>,<>,<>> ...

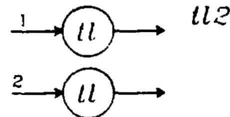
2.4. An equality predicate

Our first Marigold program will define an equality predicate on natural numbers. Since the only primitive, Boolean-valued function so far defined as a Marigold operator is the test for a null sequence, the equality test must be constructed to make use of ***null?***.

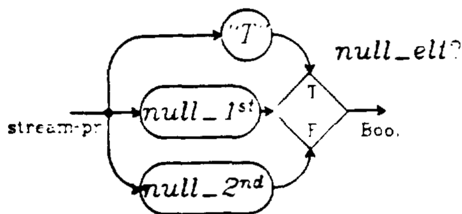
The program given below is a composition of three parts. The first segment takes the original argument, a pair of streams representing numbers, into a stream of such pairs. The members of each pair are the tails of the respective stream elements of the preceding pair. The second graph component represents a μ-operator that tests each element of this sequence of pairs, to determine whether one or the other is the null stream (the number zero), and delivers the stream of pairs that meet this test. The third component converts this stream of pairs into a stream of Boolean values, T if both elements of a pair are null, F otherwise. Finally, an application of the element selector ***hd*** extracts from the stream of Boolean values the first one.

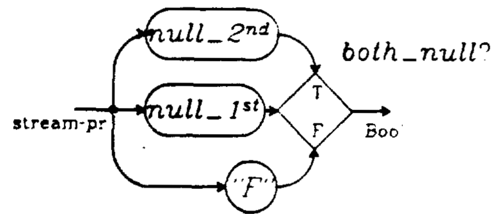


where *tl2* is



and the two comparison predicates are:



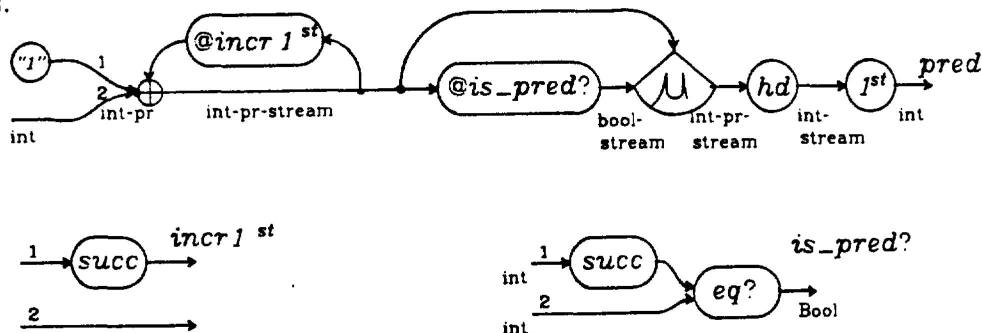


2.5. Predecessor

A predecessor is a number whose successor is equal to the given argument. In order to form a predecessor from the basis with which we have started, we first construct a sequence of

numbers, starting from zero, then compare the successor of each one with the argument. In order that the argument can be presented to a sequence of comparisons, it must be copied into the elements of a stream. The first segment of the program generates an infinite sequence of pairs, whose first elements are numbers ascending from zero, and whose second elements are copies of the given argument.

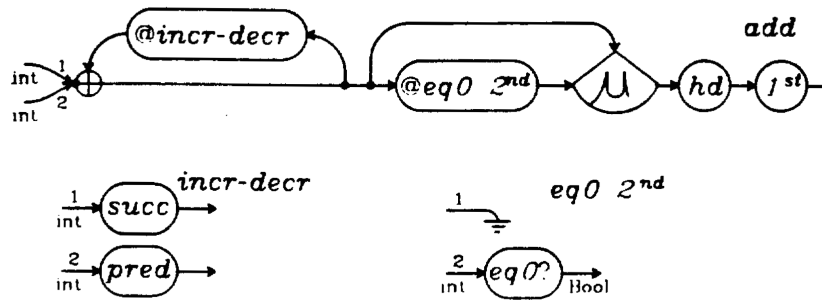
The first program segment is composed with a second, which selects from the infinite stream just those pairs for which the successor of the first element is equal to the second, if any such pairs exist. Finally, by selecting the initial pair from the selected sequence, and taking its first element, we obtain a number that is the predecessor of the original argument, if any such number exists.



In this example, it is not difficult to prove (from a suitable axiomatization of streams) that the program will yield a result when applied to any stream object except the null stream.

The example as given above does not rely upon the particular representation for numbers chosen in Sec 2.3. It uses only the constant 1 and the functions *succ* and *eq*?. Of course. a simpler realizations of ***pred*** is possible by taking account of the representations of numbers as streams. It is just

Continuing the development of the elementary arithmetic functions, the next is *add*. This follows nearly the identical scheme as does *pred*; the differences are that it takes a pair of arguments instead of a single argument and the constant *zero*, and that the functions *incr1*st and *is_pred*? are replaced by the functions *incr_decr* and *eq0 2*nd defined below.



The predicate *eq0?* is of course the same as *null?* when integers are represented as streams of (null) streams. Multiplication and exponentiation functions follow similar graphical schemes.

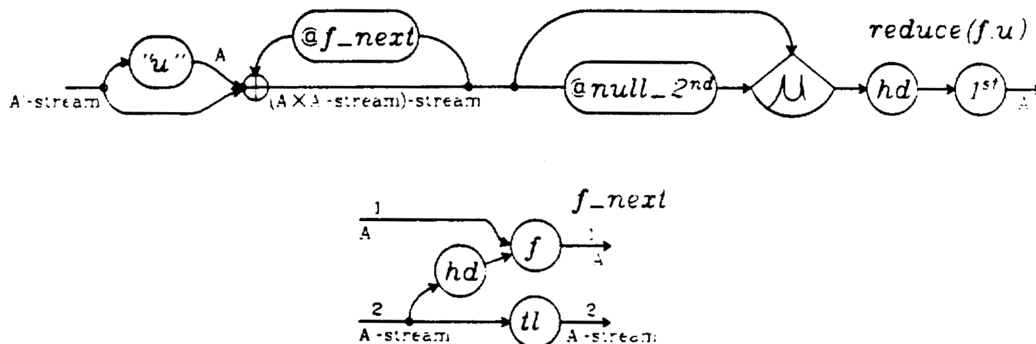
Functions defined in Marigold extend naturally to stream-valued arguments. The apply-to-all functional is the canonical extension operator. Other schemes can be programmed, however. For instance, a useful scheme is that which uses a binary operator and maps a stream into a scalar value.

2.6. A reduce operator

The program scheme used to define *pred* can also define a *reduce* operator. Let

$$f : A \times A' \to A$$

be a binary function and let $u : A$ be a constant. Then the program



reduces an argument of type *A'−stream* to a value of type A by iterated application of f, using u as a left unit. This *reduce* operator generalizes that of APL. Its unit is made an explicit, rather than an implicit argument, and the function f is not required to be associative.

Elementary arithmetic functions provide useful examples to illustrate the style of Marigold programs, but it is not surprising that they can be given in a straightforward manner using flow-graphs. Functions that are known to admit iterative evaluation

(without use of a stack) have often been used to illustrate function definition by flow-graphs, without the use of recursion. However, the next example is a function that has no tail-recursive definition.

2.7. Reverse

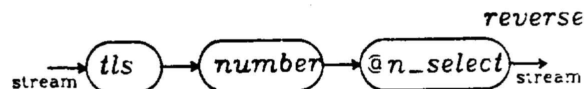
The function which reverses a (finite) stream is not so simple to define in Marigold as it is in LISP, for instance. The fundamental data type, the stream, has operators that make it natural to use as a FIFO data structure, rather than as a LIFO data structure. With a LIFO data structure, the reverse of a list is obtained just by rewriting it, one item at a time. A tail-recursive, equational definition of ***reverse*** can be defined in just this way,

$$reverse\ x = (rev\ nil\ x)$$

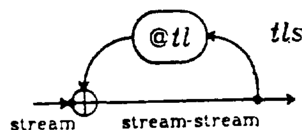
where

$$rev\ u\ v = (cond\ ((null\ v)\ u)\ (t\ (rev\ (cons\ (car\ v)\ u)\ (cdr\ v))))$$

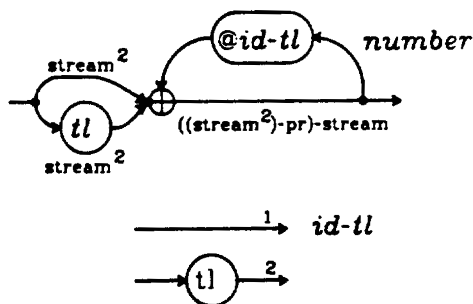
The above construction is possible because ***car*** is a function that returns the last item "*cons*ed" onto a list. With streams, the extractor function ***hd*** returns instead the first item that was entered in the stream. In the following Marigold program for ***reverse***, higher-order streams are generated, in order to permit the components of an original stream argument to be extracted in reverse order. The program is a composition of three component functions:



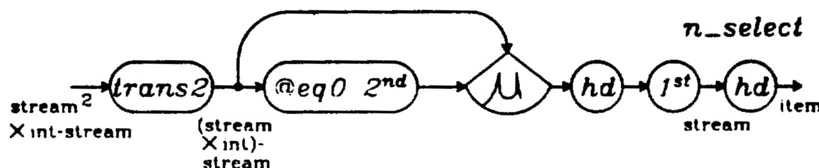
The first component function,



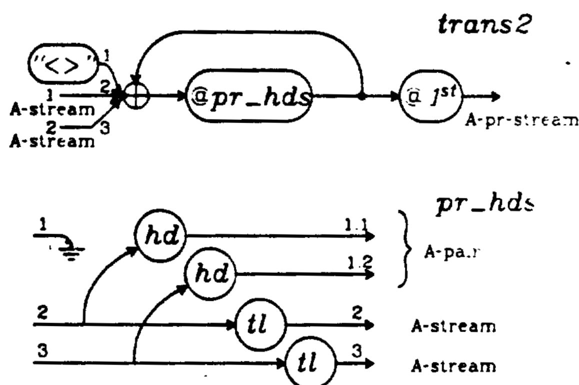
expands a stream of length l into a second-order stream which orders the terminal substreams of the argument. The second function,



pairs copies of the stream produced by ***tls*** with successive terminal subsequences of itself. Each such subsequence represents a descending sequence of natural numbers, n, $n-1, \ldots 0$, where $0 \leq n < l$. The last function,



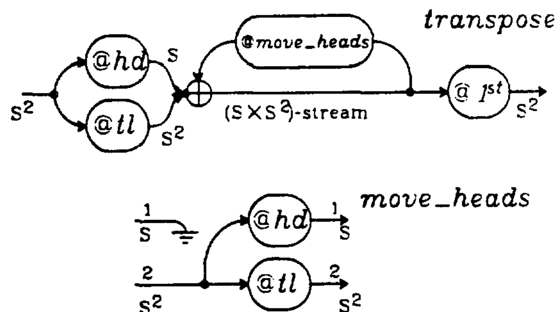
uses the number-sequence in the second component of its argument pair to control the selection of a component item-stream from its first argument. Taking the heads of the selected item-streams yields the reverse of the item-stream that was presented as the argument to *reverse*. The function *trans2* is the transpose of a stream-pair; it maps a pair of streams into the single stream whose elements are the pairs of elements of corresponding index from the argument streams.



2.8. Transpose

A finite vector is just a stream of finite extent. A matrix is a finite stream of finite streams, each of the same length. The function *transpose*, given below, transposes a matrix. Its scheme is yet another instance of the scheme we first introduced to define *pred*.

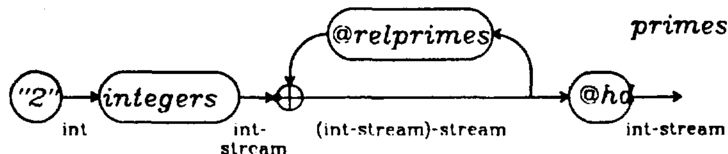
However, the iterated function, ***move_heads***, applies the stream selector functions ***hd*** and ***tl*** not to the whole argument, but to elements of a second-order stream argument.



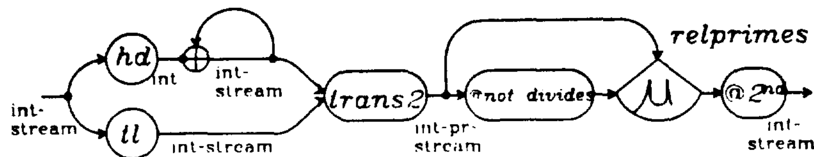
In this example, second-order streams occur as the input data. A third-order stream is created as an intermediate data representation in the iterative cycle. In the following example, higher order streams are created in lieu of higher order functions.

2.9. Prime numbers

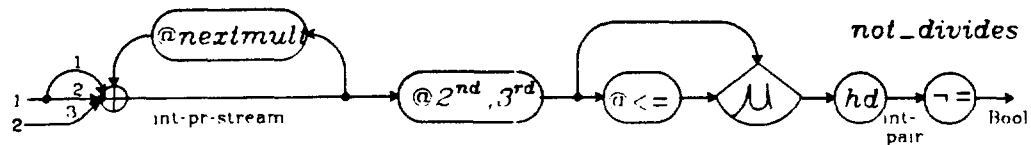
The style of programming that we have been illustrating is well-suited to the construction of a sieve for filtering the primes from a sequence of natural numbers. The algorithm presented here is that published in [HeM76] and attributed to P. Quarendon. A stream of natural numbers is generated from a seed value which is the first prime. A second-order stream is then generated by filtering from each preceding stream of numbers all those that are divisible by the first element of that stream. This is the task of the stream-to-stream function ***relprimes***. Finally, the sequence of the initial elements of each stream will be the stream of prime numbers.

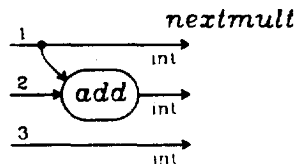


The filter that removes relative primes from a stream first pairs the head of the original stream with each of its succeeding elements. This stream of pairs is then examined to determine which of the pairs are multiples, and these pairs are removed. Finally, the first element is stripped from each pair, leaving a stream of numbers that were not divisible by the head of the original stream.



The final component performs the test of a pair of integers to determine whether or not the second is a multiple of the first. A copy is made of the first value, then a stream of triples is formed, in which every multiple of the original first value occurs. From this stream of triples is selected the first one in which the synthesized multiple equals or exceeds the value of the second number in the original pair. The final result of the test is gotten by an equality test comparing the synthesized multiple with the hypothesized multiple.





2.10. Primitive recursion

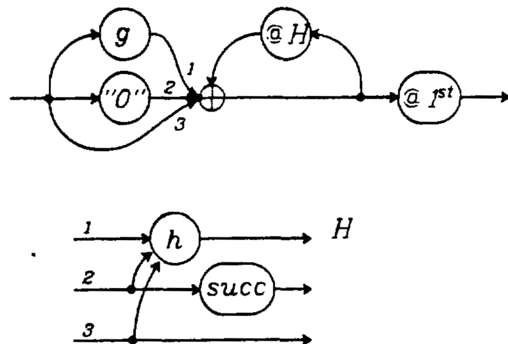
The programs we have seen follow the same schemes of stream construction and selection over and over again. These schemes also seem to illustrate a very natural programming paradigm. It is not original with Marigold. The paradigm of stream construction is directly analogous to primitive recursion as a scheme for function definition.

The primitive recursion scheme to define a function f is

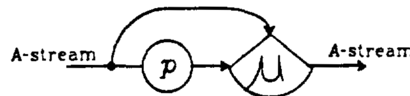
$$f(0, \bar{y}) = g(\bar{y})$$

$$f(x+1, \bar{y}) = h(f(x, \bar{y}), x, \bar{y})$$

where g and h are previously defined functions, and $\bar{y}$ is an argument tuple (one or more arguments). This scheme translates into Marigold as:



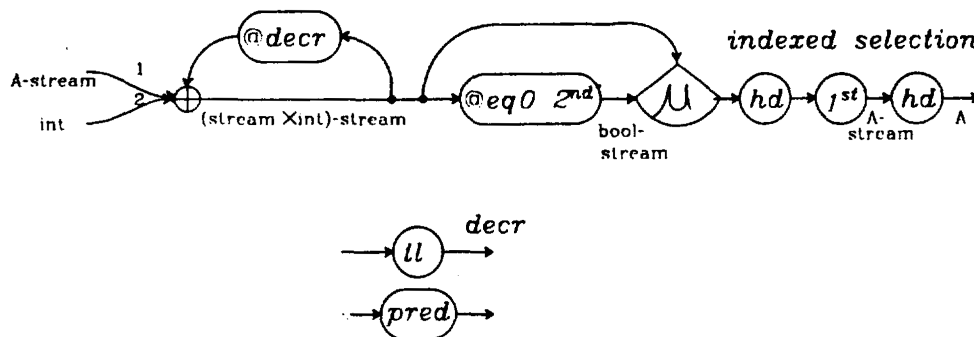
As mentioned previously, the Marigold scheme



corresponds to Kleene's μ operator. The μ operator, in combination with primitive recursion schemes and a suitable set of basis functions, comprises a mathematical language in which all partial recursive functions can be defined. Marigold also has this universal expressiveness.

2.11. Indexed selection

An important operation is selection of an element from a stream, according to the value of a natural number used as an index. The following program performs indexed selection:



In this example, if the first component of the argument is bound to a stream value S which is interpreted as the graph of a function, $f_S : A \to B$, and the second component is bound to an integer encoding n of a type A object, then the program represents the application $f_S\ n$. In the next example we shall make use of this interpretation.

2.12. Ackermann exponential

Next, we have the classical example of a total function which is not primitive recursive. The recursion scheme by which this function of three variables is customarily defined starts by giving as a primary basis the cases when the first variable is zero,

$$A(0,0,y) = y$$
$$A(0,x+1,y) = A(0,x,y) + 1$$

A secondary basis specifies the function for all values of its first argument, when the second argument is held at zero,

$$A(1,0,y) = 0$$
$$A(z+2,0,y) = 1$$

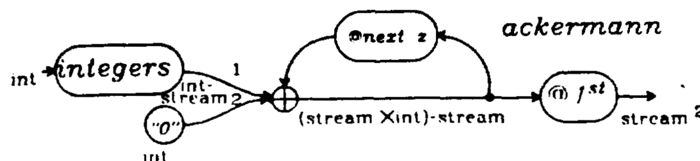
The definition is completed by the dyadic recursion equation

$$A(z+1,x+1,y) = A(z,A(z+1,x,y),y)$$

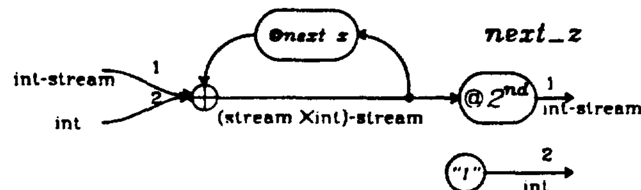
As a function of three variables, the Ackermann exponential could be represented by a stream of streams of streams of integers; for economy of notation, we represent it below as a stream of streams of integer-valued expressions in the variable y,

		$z \longrightarrow$					
		0	1	2	3	4	5
x	0	y	0	1	1	1	1
	1	$y+1$	y	y	y	y	y
	2	$y+2$	$2y$	y^2	y^y		
	3	$y+3$	$3y$	y^3	y^{y^y}		
	4	$y+4$	$4y$	y^4	$y^{y^{y^y}}$		

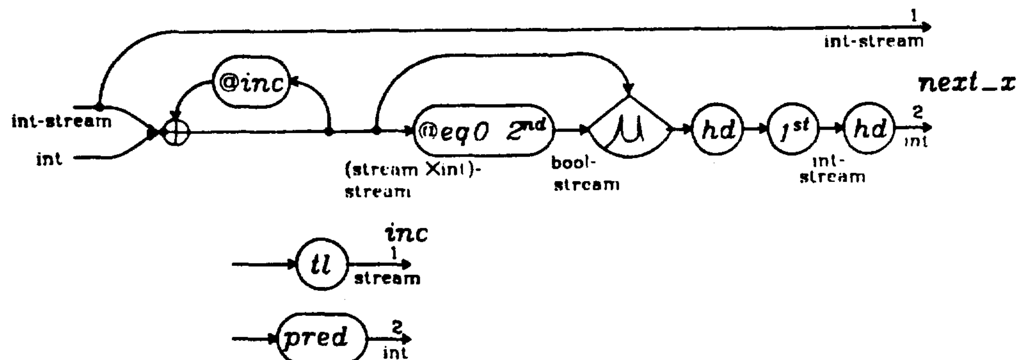
A Marigold program that generates the Ackermann exponential of three variables is presented below, in a hierarchical definition. At the first level, the two components of the input are the index and the value of the first stream (first column in the table above) of the representation of the Ackermann exponential. The remaining streams are generated by iterative application of the function ***next_z***.



The next level of program composition presents a definition of *next_z* whose inputs are components of the stream generated by the primitive recursion cycle in the program shown above. The computation of *next_z* uses the zth stream and the value of the initial element of the $z+1^{st}$ stream to generate the remaining elements of the $z+1^{st}$ stream.



The innermost level of program composition defines the function *next_x*, which generates the $x+1^{st}$ value in the $z+1^{st}$ stream. It does so by using the value of $A(z+1,x,y)$ as an index into the stream representing $<\cdots A(z,\xi,y)\cdots>$ Note that selecting from a stream, S, the value indexed by an argument, x is the same operation as application of the function f_S to a value x.



In [Nor81], the analogous phase of the construction is done by recourse to a second-order function, which carries out the application described above.

2.13. Merge

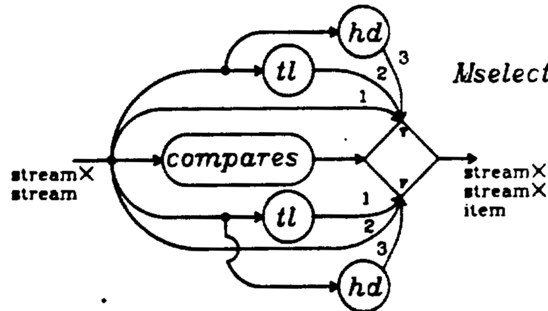
Two streams of objects from a totally ordered domain are to be merged into a single stream such that if the argument streams were presented in order, then the result stream will also be in order. Repetition of objects is allowed. This example is of interest because the customary way to specify it using recursion equations makes two recursive calls, with differing arguments:

```
merge x y =
    (cond ((null x) y)
          ((null y) x)
```

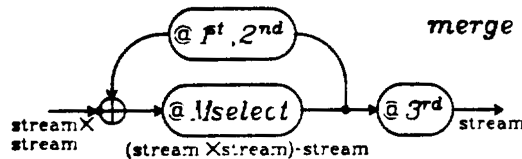
((*less* (*car* x) (*car* y)) (*cons* (*car* x) (*merge* (*cdr* x) y)
(*t* (*cons* (*car* y) (*merge* x (*cdr* y)))))

The two alternatives for the as-yet-unmerged remainders of the argument streams can be handled with a conditional selection form.

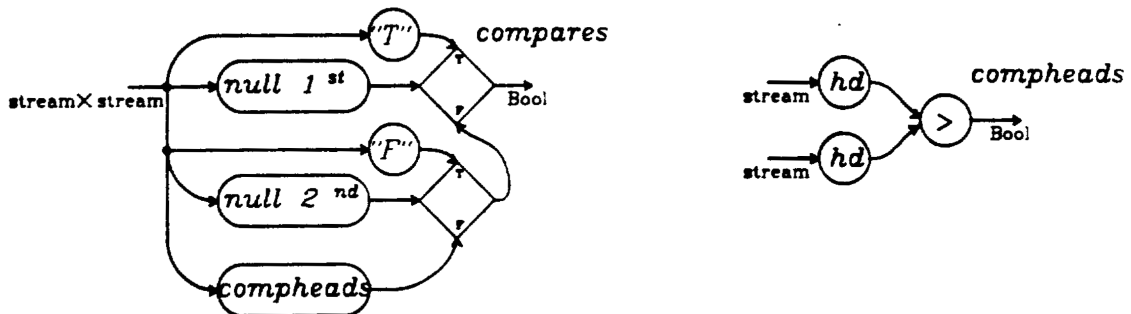
Let *Mselect* be the flow-graph



in *merge*:



where *compares* is



This flow-graph program appears somewhat more complicated than the examples that have preceded it, but it is not more complicated than is the recursion equation definition of *merge*. The algorithm requires case analysis on the component input streams because they may be finite, and the case analysis is reflected in the compound comparison predicate.

3. Semantics

Marigold program schemes are given meanings as operators in the partially-additive category $\mathbf{Pfn}_D$ [ArM80]. This is a direct semantics which assures the existence of fixpoints for flow-graphs restricted as are those of Marigold. Marigold schemes are graphical representations of Kleene recursion schemes, rather than being based upon the λ-calculus, as is FGL. Thus it is not necessary to use as complex a semantic definition as is required for the λ-calculus. In particular, reflexive domains are not required.

Unlike a language inspired by the λ-calculus, Marigold does not allow the definition of higher-order functions. It does not employ abstraction and thus has no variables – in this respect it is more like FP than like LISP. It is possible to define program schemes in Marigold, and to substitute instances of other Marigold functions into these schemes. But it is not possible to substitute functions into schemes recursively. Avoiding the recursive definition of higher-order functions achieves a profound simplification with respect to the λ-calculus.

In place of higher-order functions, first-order functions are applied to produce or extract from higher-order streams in Marigold. As we have seen in numerous examples (*reverse*, *primes*, *ackermann*, *merge*, etc.) higher-order streams are often defined specifically to create suspensions which will never be more than partially evaluated. Extraction of components from these suspended streams must necessarily occur in a different order than would be dictated by an applicative-order evaluation. It is perhaps in recognizing the power of suspended streams that Marigold achieves its novelty as a programming notation -- the programmer is not required to resort to naming and environment definition in order to achieve expressiveness.

3.1. Reflexive domains considered harmful

It is interesting to contemplate just why a semantics for the λ-calculus requires reflexive domains. In the λ-calculus every expression can be used as a function; in computing jargon we would say that there is no distinction between program and data. Since a function can be applied to itself, any domain that is to provide a model for the λ-calculus must be large enough to include its own function space. This is interesting, and to define such a domain is a technical *tour de force*. However in doing so, one builds a rather complex and opaque foundation for programming language semantics. Is this really necessary?

Consider an alternative. In Marigold, we have a programming notation in which programs are expressed by graphs and data are represented by streams of arbitrary order. The language has

universal power, but programs and data are separate. What then has happened to the "self-applied" function?

What has happened is that the mechanism for interpreting expressions as functions (intensional representation) is not implicit in Marigold, as it is in the λ-calculus. If you want intensional representation, then it is up to you as a programmer to invent an arithmetization for flow-graphs, and to construct a stream-of-streams that represents, for each natural number n, the graph of the function defined by flow-graph φ_n. Note the distinction: Marigold graphs are expressive in that a program to define any function, including a Marigold interpreter, can be given. But a connection between intensions (flow-graphs) and data (streams manifesting the graphs of functions) is not implicit in the programming language.

As a final comment, we should make clear that the cardinality of a domain required for a model of functions definable in Marigold is not smaller than that required for a model of the λ-calculus, since Marigold objects include infinite streams of arbitrary finite orders. It is easy to embed its semantic domains in $P\omega$, but not easy to see how to embed them in some universal domain of smaller cardinality.

3.2. Extensionality

Since the semantics of each Marigold flow-graph are given directly, rather than with respect to an environment, function definitions have the property of extensionality. Extensionality means that a definition always has the same meaning as a mathematical function regardless of context. This is an important and desirable property of a notation for programming, and one that is absent, in general, when functions are denoted by expressions in the λ-calculus. It holds of such expressions only when they are closed, i.e. contain no occurrence of a free variable.

Note that when names are used on function nodes in Marigold graphs, as they have been in the examples of Sec. 2, that the names are not free variables. Names denote specific (i.e. constant) sub-graphs, and meanings are given to fully instantiated graphs.

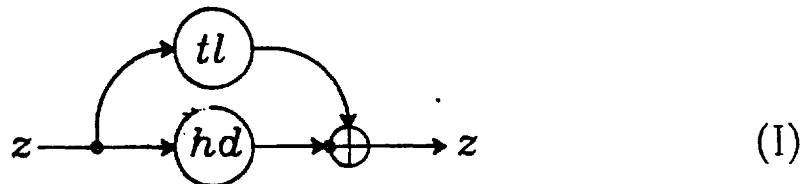
The extensionality property also holds for functions represented as streams in Marigold, for the stream is a manifestation of the trace of a function. Of course, extensionality also implies that there can be no computable equality predicate on functions, just as there can be no computable equality on infinite stream objects. Equality has not been included as a primitive predicate in Marigold. It can be defined in the language for any algebra whose carrier is a set of finitely-presentable objects, such as the integers or the rationals, but not the reals.

3.3. Semantics of flow-graphs

Partially additive categories have been studied and elaborated [ArM80] in order to provide a semantic foundation for programs, and more specifically, for programs specified by flow-graphs. Their development was motivated by the earlier work of Calvin Elgot [Elg72, Elg75].

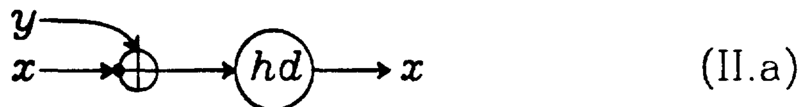
The basic idea is that the meaning of a program can be expressed as a "sum" of the meanings of the distinct paths through the program flow-graph. The set of distinct paths must be countable, but may be infinite, as will ordinarily be the case if a flow-graph contains cycles. A requirement for use of this method is that data-flow paths are determined uniquely by the program flow-graph and its input data. The theory of partially additive monoids, generalized to categories, has made these notions precise. The theory guarantees the existence of unique fixpoints to recursion equations that are expressed directly in terms of flow-graphs.

Without recapitulating the theory of partially-additive categories, let us give the axioms satisfied by the principal operators of Marigold which justify Marigold schemes as such a category. The following axiom

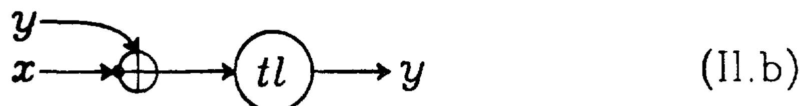


(I)

is an instance of what is called a *compatible sum axiom* in [ArM80]. It establishes that $\oplus$ is a partially-additive operator. A second axiom,



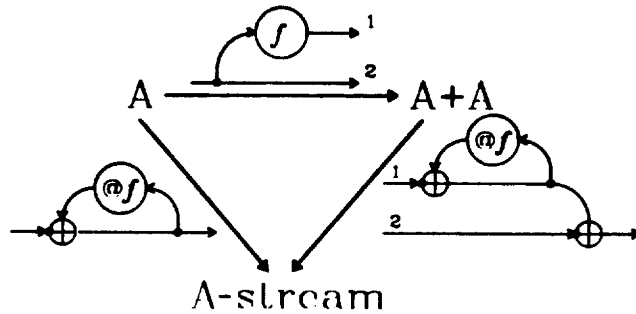
(II.a)



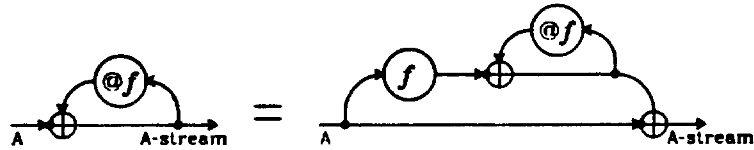
(II.b)

is an instance of what is called the ***untying axiom***. These are the only axioms of a partially additive category. We shall later give analogous axioms to justify the inclusion of the μ operator.

Let us illustrate next the flow-graph equation of the Marigold primitive recursion scheme, and describe its fixpoint. Triangle diagrams of this particular sort are called *Elgot iteration diagrams*.



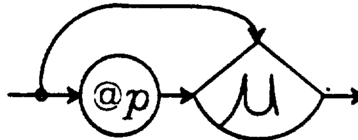
You can read this diagram as the equation



Its fixpoint is the function which, when applied to a value a, yields the infinite stream

$$<a, f(a), f(f(a)), f(f(f(a))), \cdots f^i(a), \cdots >$$

This value is well defined in the object domain of Marigold semantics.

The following graph 

represents a projection function $\mu(p)$: A-stream $\longrightarrow$ A-stream. However, when p is a recursive predicate, it also defines a complementary projection, $\mu(\neg p)$. The two projections satisfy the axioms

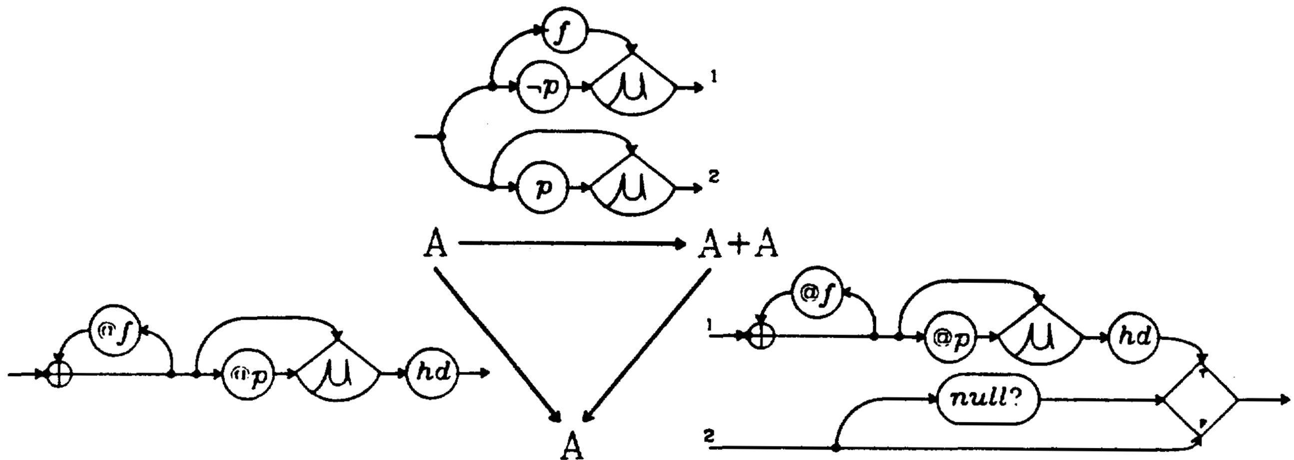
$$\mu(p) \cup \mu(\neg p) = \mathbf{id} \qquad \text{(III.a)}$$

$$\mu(p) \cap \mu(\neg p) = \bar{\phi} \qquad \text{(III.b)}$$

Axiom III.a is the compatible sum axiom and III.b is the untying

axiom for the μ-operator of Marigold.

The following diagram

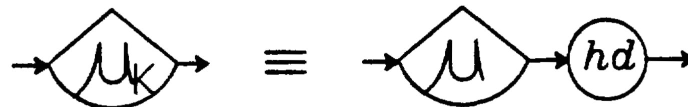


has a fixpoint which is the function that, when applied to an A-type value, a, yields the first component $f^n(a)$ for which $p(f^n(a))$ is true from the A-stream

$$<a, f(a), f(f(a)), f(f(f(a))), \cdots f^i(a), \cdots >$$

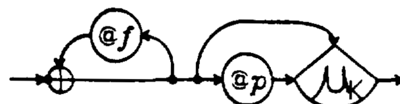
If there is no such component, then of course the fixpoint is a partial function undefined on the argument a; it yields $\perp_A$. Similarly, in case p is undefined on some element $f^i(a)$ of a stream, for any index $i <$ least n for which $p(f^n(a))$ holds, then the fixpoint is a function undefined on a.

Finally, we offer a theorem (the proof is omitted here) which relates the Marigold μ-operator, which is a projection from a stream to a substream, to Kleene's μ-operator, which is a projection from a stream to an element of the stream. Kleene's operator has an exact counterpart in Marigold; let

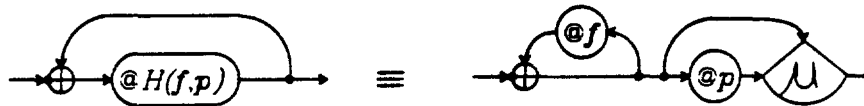


Using this abbreviation, we can state the following

Theorem: Let $H(f, p)$ be



Then



1. Evaluation of Marigold graphs

Marigold flow-graphs are evaluated by graph reduction. Graph reduction is a demand-driven, or lazy evaluation strategy in which streams are suspended until a component is demanded, then evaluated component-by-component in response to demands. Any graph, once bound to argument values, is an expression that stands for a value in the carrier set of an implementation algebra, if in fact it represents any value at all. Evaluation may be thought of as a process that replaces a graph by the value that it stands for. Under replacement of component graphs by values, no component graph is ever evaluated more than once; i.e. there is full value-sharing in evaluation by graph reduction.

Graph reduction of Marigold graphs must necessarily be demand-driven because the semantics of Marigold include infinite sequences; recall that the primitive recursion scheme applied to finite arguments has only infinite streams as models. However, the task of programming is made easier by the fact that one does not always have to worry about the finiteness of objects; in fact the only way to obtain a non-terminating evaluation is through use of the μ-operator. If a μ operator is invoked on an infinite stream to produce the next element that satisfies a predicate p, when in fact no element of the stream satisfies the predicate, then the search will not terminate.

There is reason to conjecture that it will be possible to obtain highly efficient evaluators for Marigold programs. As the reader may have noticed, the program schemes obtained are inherently iterative; there is no need to "discover" instances of tail-recursion and translate them to iterative form for the evaluator. The complexities of evaluation have entirely to do with managing suspended streams efficiently. This is a topic that has not been as extensively investigated as it should be, and I hope that the attractiveness of Marigold as a programming notation will help to motivate research into better implementation schemes for dealing with suspended objects.

References

[ArM80]

Arbib, M.A. and Manes, E.G., Partially additive categories and flow-diagram semantics, **J. Algebra 62**, 1 (Jan. 1980), 203-227.

[ArM82]

----------, The pattern-of-calls expansion is the canonical fixpoint for recursive definitions, *J.A.C.M. 29*, 2 (Apr. 1982), 577-602.

[Bac78]

Backus, J.W., Can programming be liberated from the von Neumann style? A functional style and its algebra of programs, **C.A.C.M. 21**, 8 (Aug. 1978), 613-641.

[Elg72]

Elgot, C.C., Remarks on one-argument program schemes, *in* "Formal Semantics of Programming Languages" (R. Rustin, ed.), Prentice-Hall Inc., Englewood Cliffs, N.J., 1972.

[Elg75]

-----------, Monadic computation and iterative algebraic theories, *in* "Proceedings of Logic Colloquium '73" (Rose and Shepherdson, eds.), North-Holland, Amsterdam, 1975.

[FrW76]

Friedman, D.P. and Wise, D.S., CONS should not evaluate its arguments, *in* "Automata, Languages and Programming" (Michaelson and Milner, eds.), Edinburgh Univ. Press, 1976, 257-284.

[Hec77]

Hecht, M.S., "Data-Flow Analysis of Computer Programs", American Elsevier, New York, 1977.

[HeM76]

Henderson, P. and Morris, J.H. Jr., A lazy evaluator, *in* Proc. of 1976 Sympos. on Principles of Programming Languages, ACM, New York, 1976, 95-103.

[Hen80]

Henderson, P., "Functional Programming", Prentice-Hall International, London, 1980, pp. 232-234.

[Kel79]

Keller, R.M., Semantics and applications of function graphs, Dept. of Computer Science, Univ. of Utah, Aug. 1980.

[Nor81]

Nordström, B., Programming in constructive set theory: some examples, *in* Proc. of 1981 Conf. on Functional Programming and Computer Architecture, ACM, New York, 1981, 141-153.