Systems Research Vol. 2, No. 4, pp. 285–295, 1985.
Printed in Great Britain.

0731-7239/85 $3.00+0.00
Pergamon Press Ltd.

Distributed Computation by Graph Reduction

ROBERT M. KELLER

Department of Computer Science, University of Utah, Salt Lake City, UT 84112, U.S.A.

(*Received for publication* 5 *June* 1985)

Key Words—Functional programming; graph reduction; parallel computation; distributed computation; lambda calculus; higher-order functions; Lisp; closures; equational language; data-flow.

Abstract—We describe the use and distributed implementation of a functional language based on the graph-reduction concept. Attention is given to the issue of implementing higher-order functions and concurrent evaluation. Prospects for optimization of this implementation are pointed out.

1. INTRODUCTION

THE ADVENT of practical large-scale parallel computers within the current decade is placing new intellectual demands on scientists, engineers, and programmers who must develop applications for such machines in order to speed up execution time. Difficulties of managing timing dependencies and algorithm decomposition are added to the already-formidable task of creating logically correct computer programs. Computer scientists are endeavoring to devise appropriate language tools to make such problems intellectually manageable. The emphasis here is on structuring techniques, powerful mathematical operators, and direct correspondence between specification and performance. The present article describes one line of research in this direction. We describe an execution model for programs based on functional specifications. We hope that this mode of specification and computation will open some new avenues for the reader only familiar with Fortran-like languages.

Functional languages are attractive vehicles for distributed computing because of the natural means by which they can exploit parallel execution. Since expressions are evaluated for their *value* rather than for their *effect*, expressions which do not contain one another can be evaluated concurrently. Moreover, further relaxation of sequentiality can be obtained by the introduction of *lenient operators*, which permit certain expressions to be evaluated concurrently even if one is contained in the other. Necessary synchronization can be invisibly enforced by the evaluation mechanism.

An important mechanism for achieving the above form of evaluation is *graph reduction*. This entails representing the program as a directed graph in which nodes correspond to functions, either primitive or user-defined, and arcs correspond to the "flow" of data between nodes. The present paper presents a specific graph reduction implementation, beginning with the language to be evaluated, on through the internal representation of programs and data.

2. A SIMPLE LANGUAGE

We describe a subset of the language FEL (Function Equation Language) [19] which has been implemented using graph reduction, and which is used to drive a multiprocessor simulator [21]. The subset will be adequate to describe the essential ideas in the implementation. Although syntactically different, FEL is a form of *lexically scoped Lisp*, and a Lisp syntax could have been used just as well. FEL differs from the lexically scoped language Scheme [1], but is similar to the equational language SASL [29], in that it has a *normal-order* semantics [5].

An FEL program is an FEL expression. The expression has a value, and that value is the "output" of the program. Since a value in FEL can be a very large data object, including an incrementally produced, but unending, *stream*, no computational generality is lost; indeed, generality is gained over conventional languages which do not have the concept of infinite data objects.

Input is accomplished by an expression which evaluates to (again, a possibly infinite) data object, which might be a pseudo-function "window" to the outside world. The existence of such pseudo-functions permits our exposition to concentrate, without loss of generality, on programs which do not require external input, but rather *generate* their output.

As an example, consider the following (intentionally unoptimized) prime-number generating program in FEL:

```
primes = keep | prime : nums_from : 2          (1)
```

```
nums_from : n = n ^ nums_from : (n + 1)            (2)
prime : n = relatively_prime : [2, n]              (3)
relatively_prime : [m, n] =
  if m > = n
    then true
    else if divides : [m, n]
      then false
      else relatively_prime : [m + 1, n]           (4)
keep | P : x = if P : first : x
    then first : x ^ keep | P : rest : x
    else keep | P : rest : x.                      (5)
```

The above symbols have the following explanation:

: (read *of*) represents the application of a function (the left argument) to an argument (the right argument). This operator has a right-associative syntax (i.e. $f : g : x$ means $f : (g : x)$).

| (also read *of*) is like :, except that the syntax is left-associative ($(f \mid g \mid x$ means $(f \mid g) \mid x)$. Also, | will take precedence over : ($f \mid g : x$ means $(f \mid g) : x$ rather than $f \mid (g : x)$).

^ (read *followed-by*) represents a function, the result of which is a stream of values, beginning with the left argument and continuing on with the stream of values which is the value of the right argument. This is an example of a *lenient* data constructor, in that the second argument need not be completely evaluated for the result to be considered a value. The companion operators *first* and *rest* act to take apart the result of ^ and do so independent of the completion of either argument). Thus, first: $(a \hat{} x) = a$ and rest: $(a \hat{} x) = x$.

[..., ..., ...] represents record or tuple creation; the value is a tuple with components which are the values of sub-expressions separated by commas. (Functions with arity >1 are viewed as 1-ary functions operating on a tuple. 0-ary functions are functions operating on the null-object [ ].) In FEL, [..., ..., ...] is also a lenient operator; the function *tselect* is used to extract the *i*th argument, independent of the completion of the other arguments. That is, *tselect*: $[i, x_1, x_2, \ldots, x_n] = x_i$.

if... then... else... (abbreviated *if*) is a 3-ary function which evaluates its first argument, then yields the value of its second or third argument, depending on the value of the first.

= is equality, and may either be used to define an object or to compare two objects.

With these in mind, the above program can be read as follows:

Equation 1 says that *primes* stand for the value keep | prime : nums_from : 2, i.e. keep those numbers in the sequence 2 3 4 5 ... which are prime. *keep* is a *filtering* function, *prime* is a predicate, and *nums_from* is a *generating* function, all of which are described further below.

Equation 2 says that the (stream of) numbers beginning with n consists of n followed by the (stream of) numbers beginning with $n+1$.

Equation 3 says that a number n is prime if it is relatively prime to numbers in the range $2 \ldots n-1$.

Equation 4 says that a number n is relatively prime in the range $m \ldots n-1$ if either $m \geq n$ or if m does not divide n and n is relatively prime to the range $m+1 \ldots n-1$.

Equation 5 says that the result of keeping those items for which P holds in the sequence x is determined as follows: if P holds for the first item in x, then follow the first item by keep | P of the rest of x. Otherwise, just give the value of keep | P applied to the rest of x.

We suggest that the correctness of the above program is essentially self-evident, assuming correct implementation of the primitive functions used.

To form a complete program, we must indicate the output value (in this case, we choose "primes"). This is done by packaging the entire set of equations as an *equation group* indicated by {...}, and indicating the overall result by prefixing the word RESULT:

```
{RESULT primes
  primes = keep | prime : nums_from : 2
  nums_from : n = n ^ nums_from : (n + 1)
      .
      .
      .
  }.
```

Since the equations describe what is to be computed, rather than a sequence of commands, the order of listing above is irrelevant.

3. BASIC FUNCTION CODING

In this section, we describe the implementation of programs of the type described above. Each defined function is associated with a *code block*. The code block contains code for evaluating the body of the function, given bindings of its arguments. In simplest form, the code can be thought of as a *function graph*, a diagram representing the flow of data in the body expression. Each operator in the body is a node in the graph, as is each argument. In equation (2), for example, the body is $n \hat{}$ nums_from : $(n+1)$ and there

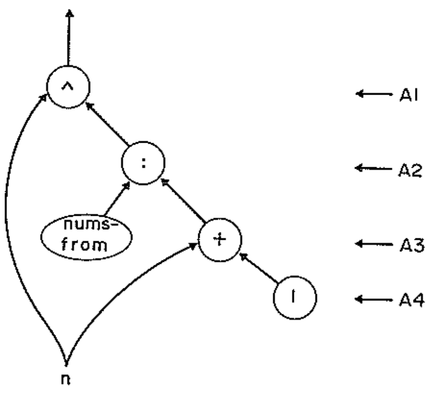

Fig. 1. Function graph for *nums_from*.

is a node for each of ^, +, and :, as shown in Fig. 1. There is also a node for the argument *n*, and a node representing the function *nums_from* which is used recursively. This node evaluates to the code for the *nums_from* function itself.

By convention, each node has only one output arc, and we think of the node as corresponding to this arc. A node may have several input arcs, and these correspond left-to-right with the arguments of the function in the node.

In providing a linearized representation of such a graph, suitable for machine evaluation, each node and its corresponding arc is given a *symbolic label*, either the name of the argument or function being applied, or an artificially generated label such as A1, A2, A3... With this in mind, the code block for the function *nums_from* can be shown as

nums_from:

label	operator	operands
	RESULT	A1
n	ARGUMENT	
A1	^	*n* A2
A2	APPLY	nums_from A3
A3	+	*n* A4
A4	NUMBER	1
nums_from	CLOSURE	nums_from

Most symbolic operand names are translated into *relative offsets* when the code is assembled. One exception is the first operand of the CLOSURE operator. This operand is instead translated to a *global address* of the code block for the function named, in this case *nums_from* itself. Thus, one purpose of CLOSURE is to "transform" the global address *nums_from* into something useable locally.

Let us consider the code block for the function *prime*. This would be translated as shown below. Here CONS is an operator which creates a *tuple* with one component for each argument, and is treated in the same manner as ^ above, except that it can have arbitrarily many arguments. (CONS is the name for the constructor [..., ..., ...] described earlier.)

prime:

label	operator	operands
	RESULT	A1
n	ARGUMENT	
A1	APPLY	relatively_prime A2
A2	CONS	A3 *n*
A3	NUMBER	2
relatively_prime	CLOSURE	relatively_prime

4. EVALUATION

The following describes the basic idea of evaluation by graph reduction. Again, we begin with an unoptimized version for simplicity.

An important property of functional languages is that of *referential transparency*: an expression can be replaced with another expression having the same value. This underlies the basic *driving mechanism* for graph reduction; by replacing expressions with others having the same value, but which are *closer* to the ultimate value in some appropriate sense, we eventually converge to something representing a useable answer.

In terms of storage, we would like to view a node of the graph as representing the expression rooted at that node, and replace it with the corresponding value. For example, in *nums_from*, the node for expression $n+1$ is rooted at the + operator node, and the latter can be replaced with the value of $n+1$ once *n* is known. In general, the act of evaluating such an expression will cause the evaluation of other expressions, which in turn are replaced by their values. A more detailed presentation will be given in Section 11.

The above form of evaluation is obviously *destructive* of the code graph. It points up the need to make a *copy* of the body of the function before proceeding with the evaluation. This copying is done as the result of an application of the function to an argument. Accompanying the copying is the *linkage* of both the result and argument of the function being applied to the copy of the body. For example, each

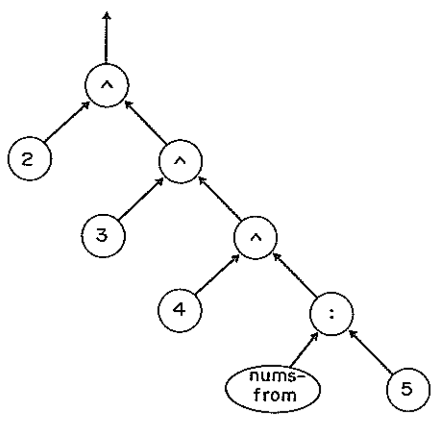

Fig. 2. Expansion of nums_from : 2.

recursive call of *nums_from* would result in a new copy of the body, and a new set of linkages. Figure 2 shows how successive calls of *nums_from* correspond to expansion of the graph for its argument. The result is a spatial representation of the stream of numbers, beginning with the argument of the first call.

We have described above the conceptual elaboration of the graph. The driving force of the elaboration is the *demand* for successive components of the stream, as would be done by *first* and *rest* operators; the ith component of stream x is demanded by select: $[i, x]$, where

```
select : [i, x] =
   if i = 1
      then first : x
      else select : [i−1, rest : x]
```

which gives

$$\text{first}: \underbrace{\text{rest}:\text{rest}:\ldots:\text{rest}}_{i-1\text{ times.}}:x$$

Only the demanded portion of the graph is elaborated, which is the reason why we can deal effectively with conceptually infinite data objects using finite resources.

5. IMPORTED VALUES

Another important aspect of graph reduction is the means for implementing *imported values*. These are used to impart a lexical or "block" structure to the language, and are the key to the realization of "higher-order" functions.

As an example, consider the pair of equations for *prime* and *relatively_prime* (equations 3 and 4). It is noted that the value of n in *relatively_prime*, once introduced at the outer level, remains unchanged for each inner call (although it varies among outer calls, of course). We could take note of this fact by using an alternate definition of *prime* in which *relatively_prime* is invisible:

```
prime : n =
      {
      RESULT relatively_prime : 2
      relatively_prime : m =
         if m > = n
            then true
            else if divides : [m, n]
               then false
               else relatively_prime : (m + 1)
      }.
```

Here we have "factored out" the argument n of *relatively_prime*; instead, n is an "imported value" (also called "free variable") which is determined by n in the particular call of prime. Thus, *relatively_prime* actually represents a different function for each call of prime.

Such imported values are handled in the following way: The CLOSURE operator takes one additional argument for each import. These are collectively packaged into a *tuple* which, along with the code block address, forms a *closure* data object. This provides a complete representation of the function, with constant-time access to any imported values. Unlike the first argument to CLOSURE, which is always the global name of a code block, these additional arguments are given by local symbolic addresses. The ramification of minimizing the number of global addresses is discussed further in Section 7. The code block for the function correspondingly provides symbolic labels for these imported arguments using the IMPORT pseudo-operator, which is similar in use to the ARGUMENT pseudo-operator. For example, the code for this revised version of *prime* would be as follows, with n which is the argument of CLOSURE corresponding to n which is the IMPORT in *relatively_prime*:

prime:

label	operator	operands
	RESULT	A1
n	ARGUMENT	
A1	APPLY	relatively_prime A2
A2	NUMBER	2
relatively_prime	CLOSURE	relatively_prime *n*

relatively_prime:

label	operator	operands
	RESULT	A1
m	ARGUMENT	
n	IMPORT	
A1	if	A2 A3 A4
A2	> =	*m n*
A3	BOOLEAN	true
A4	if	A5 A6 A7
	.	
	.	
	.	

When the node containing the CLOSURE operator is demanded, a binding of the import values takes place. The value of the CLOSURE operator is a *descriptor* of the corresponding closure object. Thus, it is actually closure-descriptors, rather than just code blocks, which are used as the operator argument to the *apply* operator; closures are the true encoded function values.

6. HIGHER-ORDER FUNCTIONS

The clean manner in which function values are handled, as described above, permits the easy introduction of "higher-order" functions: functions which take functions as values or produce functions as results.

Since a closure is a *bona fide* value, it can be returned as the result of a function just like any other value. Consider the definition of *keep* in equation (5). This function has many uses outside the example shown, and there might be occasion to supply only the *first* argument to *keep* and use the resulting "function of *x*" as a value in other ways. In other words, *keep* is a "Curried" function; its arguments can be supplied one at a time, beginning with the innermost one *P*; keep | *P* is that function, call it *keep1*, which, with argument *x*, is the sub-sequence of items *y* in *x* for which *P* : *y* is true. To see how the function-return mechanism works, we re-write the definition of *keep* to emphasize its "two-tiered" nature:

```
keep | P =
      {
      RESULT keep1
      keep1 : x = if P : first : x
        then first : x ^ keep1 : rest : x
        else keep1 : rest : x
      }.
```

We now demonstrate how such a function is implemented in terms of code blocks. First consider the definition of *keep*. It has one argument, and returns the function *keep1* which has *P* imported.

keep:

label	operator	operands
	RESULT	A1
P	ARGUMENT	
A1	CLOSURE	keep1 *P*

The result returned is a closure, with code corresponding to *keep1* and import corresponding to *P*. Now we see how this links with the code for *keep1*:

keep 1:

label	operator	operands
	RESULT	A1
x	ARGUMENT	
P	IMPORT	
A1	IF	A2 A3 A4
A2	APPLY	*P* A5
A3	^	A5 A4
A4	APPLY	A6 A7
A5	FIRST	*x*
A6	CLOSURE	keep1 *P*
A7	REST	*x*

7. STRUCTURE-PRESERVING COPYING

A subtle point of the application process (also called "beta reduction"), noted by Wadsworth [31] and independently implemented by our method, is in the manner in which bodies are copied, which should be "*as a graph, not as a tree*". That is, any sharing of nodes in the body must be *copied* as shared nodes. The obvious recursive copying method, e.g. in FEL:

```
copy : exp =
    if atom : exp
      then exp
      else cons : [copy : car : exp,
                   copy : cdr : exp]
```

copies an acyclic graph to a *tree*, without preserving any sharing of nodes. (This code assumes, for simplicity, that the graph is constructed only of *cons* pairs, the selectors for which are *car* and *cdr*.) This method does not work at all if the graph is cyclic. The reason that preservation of sharing is so important is that without it, distinct, but equivalent, nodes will be evaluated separately, which will cause replication of work when evaluation occurs. In the case of some recursions, the attendant multiplicative effect here would deal a fatal blow to efficiency.

To preserve sharing (and to copy cyclic structures) some form of "graph marking" might be used. That is, as each node in the body is copied, it is (temporarily) marked so that it is not copied a second time, but rather the first copy of the node used. If the *location* of the node's copy is also used as the mark, then sharing is achieved by refusing to copy a marked node, but instead returning the location itself.

Considerations such as those discussed above would seem to impose a fairly expensive copying algorithm (not at all trivial to implement), necessitating the construction of a relocation-table, etc. It is here that our *code block* method contributes. We copy the body into contiguous locations, so that the relative distance between any pair of nodes in the copy is precisely that in the original. In addition, references from one node to another *inside* the body are *relative*. Thus, no code relocation is necessary. The body is literally copied location by location. In effect, the code block serves both to define the body of the applied function and as "scaffolding" to enable the preservation of shared sub-structure during the copying process. Below we show the relative addresses within the block and how they correspond to the symbolic labels in the case of *nums_from* (Equation 2). Parenthesized labels indicate symbols that are not really present. The symbol *nums_from* in the CLOSURE operator is similarly translated to a global address for this code block.

nums_from: label	operator	offsets	operands
	RESULT	.+2	(A1)
(*n*)	ARGUMENT		
(A1)	^	.−1 .+1	(*n* A2)
(A2)	APPLY	.+3 .+1	(nums_from A3)
(A3)	+	.−3 .+1	(*n* A4)
(A4)	NUMBER	1	
(nums_from)	CLOSURE	nums_from	

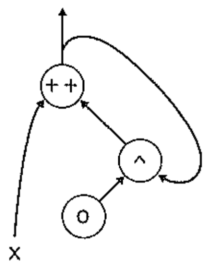

Fig. 3. An *integrator* for a stream input.

8. CYCLIC STRUCTURES

The copying technique described above can also be used to copy graphs containing cycles. Consider the example in Fig. 3, which is the body of a function which *integrates* its input stream, in the sense of computing the successive partial sums of the input. This graph employs the function ++ which adds together two streams component-wise. The first item of the input stream x is added to 0 to form the first item of output, which is then added to the next item of the input stream to form the next item of output, etc.

As a more elaborate "cyclic" example, consider the following optimized version of the *primes* program. In testing for relative primeness, we do not need to test for divisibility by all numbers from 2 up to the candidate. We only need to test for divisibility by prime numbers, since some number divides another if the prime factors of the former number do. Furthermore, we need only test for divisibility by prime numbers up to the square root of the candidate, since if a number larger than the root divides n, then the quotient is a number smaller than the root which also divides n. Thus, we could replace the definition of prime by

```
prime : n =
        {
        RESULT relatively_prime : primes
        root = sqrt : n
        relatively_prime : nums =
           if first : nums > root
              then true
              else if divides: [first : nums, n]
                 then false
                 else relatively_prime : rest : nums
        }.
```

Here we have replaced the argument to *relatively_prime*, formerly a single number, with a sequence of numbers. We now test for divisibility by successive members of this sequence, which will be *primes*, rather than by successive numbers in the natural number sequence, stopping no later than the square-root of n. This obviously is a substantial improvement in execution time.

For the present discussion, the interesting point of this example is the essential use it makes of an *induced cycle* in the underlying graph structure: prefixes of the sequence *primes* are used *while* that sequence is being computed. We must *seed* the prime sequence with its first member, 2, so that the definition becomes

primes = 2 ^ keep | prime : nums_from : 3.

Luckily, testing a candidate number for primality does not use the primes greater than or equal to the root of the number, so the cyclic definition is safe (for example, the root of 3 is less than 2).

Let us look at the code block structure now. Since primes is used in the definition of prime, it must be imported to it, so the code block for *prime* becomes:

prime: label	operator	operands
	RESULT	A1
n	ARGUMENT	
primes	IMPORT	
A1	APPLY	relatively_prime primes
relatively_prime	CLOSURE	relatively_prime

The cycle in this case is created by virtue of the fact that the imported stream *primes* is defined in terms of *prime*, which imports *primes*. The execution of this program by graph reduction has verified that each prime is computed only once.

9. REPRESENTATION USING FUNCTION-GRAPH ENVELOPES

The structure of programs such as the previous "prime" example can be described using the concept of *function graph envelopes* [15]. One important ingredient not present in the many popular variants on this basic idea [1, 9], is a graphical representation for function-valued objects which does not rely on

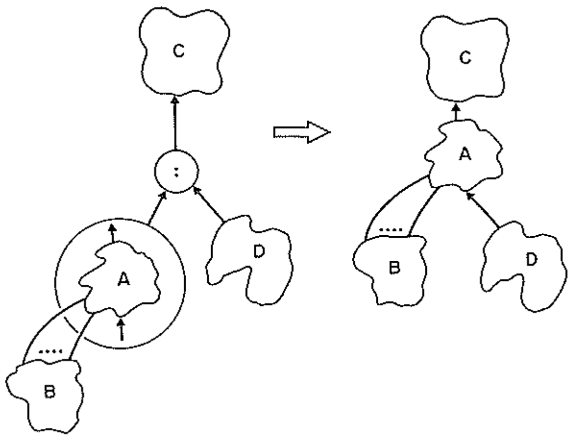

Fig. 4. Graph reduction of *apply* operator.

labels. This is done without resorting to labels by the concept of *envelopes*.

The basic setup for an *apply* operator is that there are two arguments, one a function, which is represented by a node containing the *graph* for the body of that function, with the argument and result arcs unbound. The second argument to apply is the argument of that function.

When the result of the apply operator is demanded, the apply node is conceptually *replaced* with the graph for the body, splicing in the argument and result arcs (Fig. 4).

This idea is extended to the case of functions with

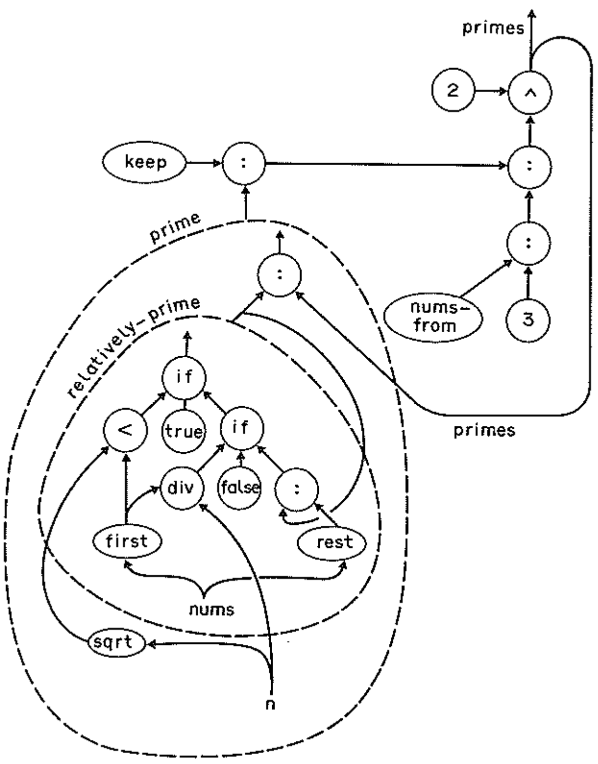

Fig. 5. Function graph for *primes*.

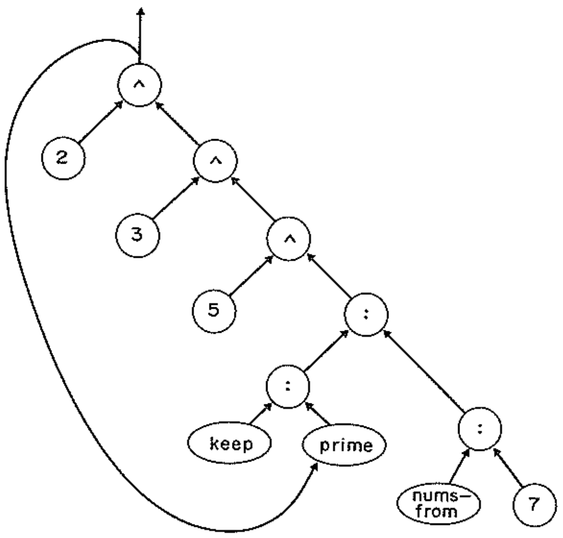

Fig. 6. Expansion of the *primes* function graph.

imports (shown as B in the figure); the latter are represented as arcs entering through the boundary of the node containing the function body. These arcs thus represent *values* which are bound by the *context* of the body, rather than when the function is applied. This gives the language its lexical scoping character. The arcs "travel with" the function body wherever the function may be sent as a value.

For the primes example, with the "cyclic" use of primes, the entire program is shown graphically in Fig. 5. It is emphasized that the *labels* on various *arcs* are purely to aid the reader in establishing a correspondence with the functional definition; they do not appear in the code block representation of the program. Figure 6 shows a snapshot of components of the primes sequence as they might develop on demand.

10. LAMBDA EXPRESSIONS

To include lambda expressions [5] in our language, we need only treat each as if it were a named function by *generating* a name for it at compile time. The notation for a lambda expression in FEL is

...vars... = >...expression...,

where vars is either a single formal variable or a tuple of variables. This designates an unnamed function which, with argument vars, yields the value of the expression with vars bound to the actual parameters in function application. The compiler generates a name, say *$lambda1*, effectively replacing the lambda expression with this name, and compiling the definition as if it were

$lambda1 : vars = ...expression....

Of course, this is done in the same {...} context in which the lambda expression appears, so that any free variables in the latter become *imports* to the code block for $lambda1.

When all functions have been compiled, the resulting code blocks form one *flat global name-space*, the names in which are replaced by pointers when the program is loaded. Note that if the same name is defined more than once in different scopes, *static renaming* is used to insure uniqueness in the global name space. For example, in the contrived definition

```
foo : x = {RESULT foo : (x + 1)
           foo : z = z}
```

there are two distinct functions named foo. The inner would be renamed at compile time by attaching a disambiguating prefix to it.

One very advantageous aspect of graph reduction is that "run-time renaming" (also called "alpha-reduction") of variables is totally unnecessary. Indeed, there are no symbolic variables at run time, as all have been replaced by relative offsets in the parameter tuple. Nonetheless, the form of a lambda expression at any stage of reduction could be recovered by retaining lists of variables and "de-compiling" the unevaluated code blocks back to expression form.

11. REDUCTION DETAILED

The following sketches the basic graph reduction algorithm. We use recursion, rather than alluding to a specific stack or "pointer-reversing" technique to maintain the discussion at a general specification level. This will be helpful in making the transition to parallel evaluation as well.

A node (representing the root of a corresponding sub-expression) in the function graph is said to be *reduced* when it is either: (1) a primitive data object, such as an integer, character, etc., or (2) a closure descriptor, containing a code-pointer and a descriptor of an import tuple, or (3) a lenient data constructor (one which does not require that its arguments be evaluated [8, 14]). Note that the *body* of a closure need not be reduced in order for the expression to be considered reduced, by this definition.

An unreduced node in the function graph is thus an application of a function to its arguments. We define a predicate *reduced* such that for any node

reduced : node

indicates whether the node is reduced. For an application node

essential-arguments : node

is the set of arguments which are required before the node can be further reduced, and

operator : node

is the operator part of that node.

The reduction algorithm is then suggestively stated as:

```
reduce : node =
  if reduced : node
    then node
    else seq[      (* do sequentially *)
      reduce : operator : node,
      for each arg in essential-args : node
        do reduce : arg,
      replace node with
        apply : [operator : node,
                 essential-args : node]
    ].
```

The difference between *args* and *essential-args* is illustrated by a node containing the 3-place operator *if* (which abbreviates if ... then ... else ...), where *if*: [*a*, *b*, *c*] is *b* if *a* is true, and *c* otherwise. Here only *a* is considered essential, and the result of apply : [*if*, args : node] above is to change the operator to either *id* : *b* or *id* : *c*, where *id* is the identity function, depending on the value of *a*. This necessitates further reduction, as indicated in the definition of reduce.

Another example is that of a lenient data constructor such as *cons* which does not demand its arguments [8, 14]. Here the set of essential arguments is *empty*, reduction taking place by operators such as *car*, *cdr*, and *tselect* which demand the arguments independently. In the reduction model, these operators are *spliced out* and their output is connected directly to the corresponding argument of the *cons* (Fig. 7).

An application of a non-primitive function may or may not have any essential arguments, depending on whether *applicative order* or *normal order* evaluation is desired [5].

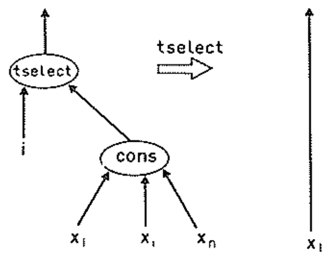

Fig. 7. The *splicing* effect in graph reduction of *select*.

The *replace* aspect of the definition is to indicate that, once reduced, the node is replaced with the reduced value, so that operators which require it in the future will not perform the reduction again. This is where sharing, which we were so intent on preserving during copying, gets exploited.

12. CONCURRENT REDUCTION

To understand *concurrent* graph reduction, note that the *foreach* statement used to reduce arguments does not specify a reduction order. Indeed, all argument reductions can be done concurrently by separate agents. To implement this effect, we can view the *foreach* statement as *enqueuing* pointers to the nodes onto some task queue, which is "served" by the agents. This is, in fact, precisely how the evaluator of AMPS [14] and Rediflow [21] work, with the task queue being distributed among many processor-memory pairs.

There are some additional necessities for the concurrent case:

1. It is necessary to be able to tell whether a node has already been demanded (enqueued) so that additional demands on a shared node will not further enqueue it. Otherwise, needless multiple evaluations would be performed.
2. It is necessary for a node, once reduced, to cause *notification* of other nodes which may have demanded its value. This is accomplished by reverse pointers (called "notifiers") from demandee to demander. Even though this technique is basically implementing a stack, it is difficult to use a pure stackbased discipline to achieve the desired effect, due to the potential demand of one value by several demanders requiring several notifications.

With the enqueuing/notification scheme, it turns out that the recursive formulation of *reduce* is replaced with a non-recursive one, eliminating the need for a stack.

The last point to be made here concerns the actual generation of concurrent activity. Let us suppose that the output of the program is a stream of values. As such, it is constructed with the lenient data-constructor ^ which does not require its arguments to be fully evaluated before the result can be incorporated in a structure. The normal evaluation mode of such an operator is a *lazy* one; the arguments are not evaluated until needed. However, this goes against the generation of concurrent activities; what we must do is *demand* the components of the output stream concurrently by temporarily imposing an appropriate "superstructure". This can be done by a recursive function *eager* which demands all components of its argument but does not wait for any of them to be complete; the result it returns is just the argument itself. The function *eager* is constructed in terms of the more primitive operator *par* [11, 16].

Reported elsewhere [21, 23] are some simulated performance figures using the approach we have described.

13. OPTIMIZATIONS AND COMPARISON WITH OTHER SCHEMES

We close by putting our graph-reduction technique into the perspective of others. First, there is the expression-interpreter evaluator introduced in Lisp 1.5 [9, 25]. The well-known difficulty with such an interpreter is that the same structural analysis must be performed repeatedly on expressions. The look-up of symbols in the environment is sequential, and hence slows down the process significantly.

Repeated analysis can be avoided by pre-compilation, such as is exemplified by the *SECD machine* approach [9, 24]. However, this approach does not maintain a suggestive correspondence between the original expression and the code being executed, nor has a parallel version of it been presented.

Next there is the work of Turner [29], who bases his evaluator on graph reduction in which nodes are primitive functions and *combinators*, such as S, K, I, B, C, etc. [4]. It is unclear to us that such combinators have any decided advantage over a lambda-calculus reducer of the type described here. Copying takes place in "block" fashion with our approach, as opposed to piecemeal with Turner's, but it nonetheless takes place in both. An advantage for debugging is that we can more readily identify our graph with the original expression than we can in the case of combinators.

Since a major role played by combinators is the "routing" of data between operators, and such routing is accomplished in our scheme by *offset addressing* in the imports tuples, the combinator scheme seems apt to introduce greater sequentiality of data access. We pose the analogy that combinators are to our representation as linked-list representations of sequences are to array representations; both can be accessed by positional index, but the linked list, due to its construction with *binary cons* (analogous to one of the routing combinators), requires linear, rather than constant, time to access values. Our representation has a sequential component, namely that a value might have to be imported through several levels of import tuples the first time it is accessed, but within an import tuple, an arbitrary number of values can be accessed directly.

An even greater access speedup over our method can be obtained using the *display* device well-known since implementations of Algol, but this is not so readily suited to implementations supporting higher-order functions, due to the great deal of display copying which would be entailed.

A major benefit of the Turner combinator approach seems to be that all operators are Curried, so that each can be used with either a full or partial complement of arguments, without special consideration. However, it is possible to adopt this convention too with the lambda-calculus approach, since Curried functions can be represented in the lambda calculus, and also since nothing precludes combinators as primitive operations.

Hughes [10] coined the term *supercombinator* to designate combinator-like (in the sense of "variable-free") objects which are of arbitrary, rather than fixed, granularity. We suggest that our code blocks have a similar property, and can be viewed as pure operators (on the pair consisting of their import and argument tuples). For this reason, we coined the term "macrocombinator" to describe them [20]. These are subject to optimizations precisely of the sort described by Hughes.

The other major point of comparison is work by the Chalmers Institute of Technology group on Lazy ML [2, 12], which is also a sequential evaluation technique based on graph reduction. Here the idea is to bypass as much of the *naive* reduction as possible by compilation of relatively mundane operators (such as arithmetic) into von Neumann machine code. For example, none of the operators in the various versions of the function *prime* entails any stream creation, so the entire graph part of this definition could be replaced with a single operator. We view this as consistent with the approach defined here; to cast the Lazy ML approach into our framework, the operators become coarse-grained functional "glue" between data constructors. The Chalmers approach has demonstrated a very high efficiency, almost making sequential functional programming competitive with conventional models. It will be interesting to see how well it can be extended to a concurrent evaluation model. We are currently exploring a stack-based technique with demand and wait operators toward this end.

The evaluation model described here differs from that in [14] in one essential way. In the latter, we used addresses relative to the *beginning* of each code block, rather than to each word itself. This meant that the code block always had to be treated as a unit, and that a pair-address was used to refer to individual words. The treatment of blocks as a unit has a resource-usage flaw in that, during garbage collection, the entire block must either be collected or left as is; this meant that a number of words could be tied up for the sake of a few words in a block.

The general problem of reduction computations consuming much storage was addressed in one study leading to the "Rediflow" concept [22, 27]. There, we introduced special stream processes (inspired by [13]) which do not require reduction for their implementation, but which interface cleanly to reduction-implemented functions. Another extension which fits nicely into the framework presented for function objects is to *permit* overt destructive modification of variable values. This is a form of so-called *object-oriented programming*, which often describes function evaluation, possibly with side-effects, as sending a *message* (the argument) to an *object* (the function) and receiving a *reply* (the result). An exposition of this style may be found in [1].

14. FURTHER READING

The reader wishing to read further on the subject of functional programming or highly parallel architectures is invited to consult the following references. The literature of engineering sciences, particularly electrical engineering and control theory, has seen many uses of graphical models for function-based systems. Consult [7] for the use of such concepts in simulation, which formed the basis of one of the earliest functional languages. The connection between this and the current model is brought out in [17].

The lambda-calculus [5] contributed to the development of Lisp [25], from which a great many functional languages, including the one presented here, have sprung. Formerly, these languages were considered the exclusive domain of symbolic computation and artificial intelligence, but there is now underway effort to bring their benefits into scientific computation as well. It is important to realize that conventional programming styles can be embedded within the graph-interconnection style, enabling combination of the best aspects of both for highly parallel computers [13, 22]. New and elegant syntactic constructs are still being found which bring the program closer to the application [3, 30]. Graphical formalisms based on functional programming have been proposed as software development tools [6, 18].

An architecture for the graph-reduction model is developed in [14, 21]. Other related architectures can be found in the various forms of *data-flow*. See [28] for a survey of concepts in this area. It is noteworthy that architectures with instruction sets optimized for functional languages are now beginning to be developed by vendors [26].

15. CONCLUSIONS

We have described an evaluation method for a functional language based on graph reduction. The handling of higher-order functions has been shown.

The method can be derived from the basic principles of the lambda calculus, being an instance and elaboration of a method suggested by Wadsworth. The method is efficient in this class, since there is no need for *renaming* of variables. All variable names are completely compiled into relative offsets prior to run-time. It has been indicated how the method is suitable for a distributed evaluator.

Acknowledgement—Work reported herein was supported by a grant from the IBM Corporation and by NSF grants MCS-81-06177 and MCS-81-21750.

REFERENCES

1. H. Abelson and G. J. Sussman, *Structure and Interpretation of Computer Programs.* M.I.T. Press, Cambridge, MA (1985).
2. L. Augustsson, A compiler for lazy ML. In *Symposium on Lisp and Functional Programming*, pp. 218–227. ACM, August (1984).
3. O. P. Buneman, R. E. Frankel and R. Nikhil, An implementation technique for database query languages. *ACM TODS* 7 (1982), 164–186.
4. W. H. Burge, *Recursive Programming Techniques.* Addison-Wesley, Don Mills, CA (1975).
5. A. Church, *The Calculi of Lambda-Conversion.* Princeton University Press, Princeton, NJ (1941).
6. A. L. Davis and R. M. Keller, Dataflow program graphs. *IEEE Computer* **15** (1982), 26–41.
7. J. W. Forrester, *Industrial Dynamics.* Massachusetts Institute of Technology (1961).
8. D. P. Friedman and D. S. Wise, *CONS should not Evaluate its Arguments.* Edinburgh University Press, pp. 257–284 (1976).
9. P. Henderson, *Functional Programming.* Prentice-Hall, Englewood Cliffs, NJ (1980).
10. R. J. M. Hughes, Super combinators: a new implementation method for applicative languages. In *The 1982 ACM Symposium on LISP and Functional Programming*, pp. 1–11. Association for Computing Machinery, August (1982).
11. B. Jayaraman and R. M. Keller, Resource control in a demand-driven data-flow model. In *Proc. Int. Conf. on Parallel Processing*, pp. 118–127. IEEE (1980).
12. T. Johnsson, Efficient compilation for lazy evaluation. In *Symposium on Compiler Construction*, pp. 58–69. ACM, June (1984).
13. G. Kahn, The semantics of a simple language for parallel programming. In *Information Processing 74*, pp. 471–475. IFIPS, North Holland (1974).
14. R. M. Keller, G. Lindstrom and S. Patil, A loosely-coupled applicative multi-processing system. In *AFIPS*, pp. 613–622. AFIPS, June (1979).
15. R. M. Keller, *Semantics and Applications of Function Graphs.* Technical Report UUCS-80-112, University of Utah, Computer Science Department (1980).
16. R. M. Keller and G. Lindstrom, Applications of feedback in functional programming. In *Conference on Functional Languages and Computer Architecture*, pp. 123–130. October (1981).
17. R. M. Keller and G. Lindstrom, Applications of feedback in functional programming. In *Functional Programming Languages and Computer Architecture*, pp. 123–131. October (1981).
18. R. M. Keller and W-C. J. Yen, A graphical approach to software development using function graphs. In *Proc. Compcon 81*, pp. 156–161. IEEE (1981).
19. R. M. Keller, *FEL* (*Function Equation Language*) *Programmer's Guide.* Technical Report 7, University of Utah, Department of Computer Science, AMPS Technical Memorandum (1982).
20. R. M. Keller, G. Lindstrom and E. Organick, Rediflow: a multiprocessing architecture combining reduction and data-flow. In *PAW 83: Visuals used at the Parallel Architecture Workshop.* Department of Energy, Office of Basic Energy Sciences, University of Colorado, Boulder, January (1983).
21. R. M. Keller and F. C. H. Lin, Simulated performance of a reduction-based multiprocessor. *Computer* **17** (1984), 70–82.
22. R. M. Keller, F. C. H. Lin and J. Tanaka, Rediflow multiprocessing. In *Compcon '84*, pp. 410–417. IEEE, February (1984).
23. R. M. Keller and G. Lindstrom, Approaching distributed databases through functional programming concepts. In *5th International Conference on Distributed Systems.* IEEE (1985), 192–200.
24. P. J. Landin, The mechanical evaluation of expressions. *Computer J.* **6** (1964), 308–320.
25. J. McCarthy *et al.*, *Lisp 1.5 Programmer's Manual.* M.I.T. Press, Cambridge, MA (1985).
26. H. Richards, Jr., An overview of the Burroughs NORMA. Burroughs Austin Research Center Memo, February (1985).
27. J. Tanaka, *Optimized Concurrent Execution of an Applicative Language.* PhD Thesis, University of Utah, March (1984).
28. P. C. Treleaven, D. R. Brownbridge and R. P. Hopkins, Data-driven and demand-driven computer architecture. *Computing Surveys* **14** (1982), 93–143.
29. D. A. Turner, A new implementation technique for applicative languages. *Software Practice and Experience* **9** (1979), 31–49.
30. D. A. Turner, The semantic elegance of applicative languages. In *Functional Programming Languages and Computer Architecture*, pp. 85–93. October (1981).
31. C. P. Wadsworth, *Semantics and Pragmatics of the Lambda-Calculus.* Ph.D. Thesis, University of Oxford (1971).