Aspects of Applicative Programming for Parallel Processing

DANIEL P. FRIEDMAN AND DAVID S. WISE, MEMBER, IEEE

Abstract—**Early results of a project on compiling stylized recursion into stackless iterative code are reviewed as they apply to a target environment with multiprocessing. Parallelism is possible in executing the compiled image of argument evaluation (collateral argument evaluation of Algol 68), of data structure construction when suspensions are used, and of functional combinations. The last facility provides generally, concise expression for all operations performed in Lisp by mapping functions and in APL by typed operators; there are other uses as well.**

Index Terms—**Compiling, functional combinations, Lisp, multiprocessing, recursion, suspensions.**

INTRODUCTION

THE purpose of this paper is to review the implications of recent results in recursive programming under a highly parallel execution environment. These are early results of a project aimed at the compilation of stylized purely recursive code. They have been presented elsewhere [5], but the implications of this type of compilation for highly parallel target code have not been gathered in one paper.

As programming tools these results appear as enhancements to applicative programming, enhancements we find necessary to strengthen classic (Lisp [21], Iswim [3] and [20]) recursive languages to express preimages of classic iterative programming techniques. While iterative programming is better developed, more familiar, and better understood than applicative programming, we strongly believe that it is unsuited to modern programming problems. Iterative programming has its roots in Turing's theoretical work. It grew with the first computers and matured through the development of programming languages (Fortran and descendants) which at first attempted to model iterative machine architecture and later, because of their universal acceptance, proceeded to determine that architecture. The work of Gödel and Church, contemporary with Turing's, supports another philosophy of programming which we feel is required to conceptualize solutions to problems for implementation on modern hardware.

We adopt a philosophy requiring all programs to be expressed as functions. There are no explicit loops (hence no goto controversy), no assignment statements [22] (only parameter bindings), and no explicit input/output functions (instead input files are taken as arguments to the main program and output files are results [9]). The language described below has been implemented semantically in a single processor environment [16]. The techniques described here do not change the semantics of the language as far as computed results are concerned. They will, however, alter a program by allowing concurrent processors to alter the space requirements as necessary to allow computation to proceed.

An issue not discussed here but implicit in all our designs is the style in which the programmer is expected to express his algorithm. Stylized recursion [5] is a methodology for formulating recursive programs which encourages good, efficient program structure and permits effective analysis and transformation before the code is executed. It is during this compilation phase that we expect that parallel processing can be specified. The programmer does not concern himself with the possibilities and pitfalls of parallelisms; the compiler selects the parallelisms from his stylized code and provides the synchronization of the processes it has identified. Our control structures allow more of this automatic parallelism selection than classical iterative control structures.

The remainder of this paper is in five parts. Only the last explicitly discusses parallelism; the first four develop a language with trivial syntactic structures but with semantics which have only been recently proposed and which allow a remarkable degree of parallelism in interpreting applicative languages. The first section introduces the elementary syntax of the language whose only control structure is a function call; an obvious parallelism allowed is collateral argument evaluation. The second feature introduced is functional combination, whereby conceptually parallel applications of several functions may be dispatched across multiple arguments yielding multiple results. Third, an extension of functional combination to arbitrary instances of the same function or the same argument allows a simple representation for the concept of "mapping" or "pipelined" operations on homogeneous structures. The fourth feature, provided by suspended argument evaluation in the primitive *constructor* function, allows for massive unstructured parallelism in a system with thousands of processors. The last section develops possible interpretations of these features at run time; the reader more familiar with parallelism than with applicative programming might scan it first in order to cast his interpretation of the four language sections in terms of something more familiar.

Manuscript received December 3, 1976. This work was supported in part by the National Science Foundation under Grants DCR75-06678 and MCS75-08145.

The authors are with the Department of Electrical Engineering, Indiana University, Bloomington, IN 47401.

0018-9340/78/0400-0289$00.75 © 1978 IEEE

THE LANGUAGE

The only structure in the language is a parenthesized acyclic list. The programmer may use it to construct arrays (e.g., a list of lists), trees, and directed ordered acyclic graphs (DOAG's). (N.B., This does not mean that the run-time structures are necessarily linear or acyclic—the compiler may have changed them.) Functions that manipulate these data may be built from a given set of elementary list operations.

Lists, represented with parentheses, are composed of elementary items or other lists. An elementary item is either an identifier (which may be bound to another value) or an integer (which is implicitly bound to itself). For example, the five following structures are legitimate as data:

```
123
FRED
(2 3 4 5 6)
( )
(FRED (8) (2 MANY ( ) (GREEN)) BANANAS).
```

A program is a function which takes as datum a list of the above sort and generates a list or an elementary item as a value. The program, however, never uses the parentheses notation explicitly.

The first programming notation is angle brackets: a bracketed sequence evaluates the list of the evaluated items of the sequence in order. For example, ⟨6 5 4 3⟩ evaluates to (6 5 4 3). Let *x* have the value (2 4 6 8) and let *y* have the value (B A N A N A S). Then ⟨*x y*⟩ evaluates to

((2 4 6 8)(B A N A N A S)).

Bracketed sequences are provided only for creating lists of fixed size and therefore they can be associated with record structures of other languages. There is also a list building function, cons, for building lists of undetermined length; but before introducing it we must introduce the syntax for function invocation.

Function invocations are represented by a pair of items separated by a colon *f*: *l*. The function position, here denoted by *f*, indicates the operation to be performed upon the argument list *l*. Combined with angle brackets this functional syntax is very suggestive of standard mathematical notation. Instead of min (*i*, *j*) we write min: ⟨*i j*⟩, and sum: ⟨2 3 4 5 6⟩ evaluates to 20. (See also [1] and [13] for similar applicative expressions.) With the binding of *x* from above, sum: *x* evaluates to 20; this case illustrates that the argument list need not be explicitly bracketed although it usually is.

A most important primitive is cons; it takes two arguments, an item and a list, and returns the list whose first element is that item and whose remainder is the original list. Thus cons: ⟨2 *y*⟩ evaluates to (2 B A N A N A S). Two complementary operations, first and rest, return the first item on a list and the list without the first item, respectively. Thus, first: ⟨*x*⟩ evaluates to 2 and rest: ⟨*y*⟩ evaluates to (A N A N A S). The semantics of these three functions are particularly interesting [8], and we shall return to them in the next section.

We shall use other elementary functions without definition; their meaning is obvious from context. These are often arithmetic, like sum, and include simple predicates: null tests if its argument is an empty list and zero tests if its argument is 0. Example functions are presented by relating a prototype invocation to its definition in terms of a conditional expression. This definition is presented as an alternating sequence of tests and values whose interpretation is assisted by the insertion of the "commenting words" if, then, elseif, and else. For example,

```
min: ⟨i j⟩ ≡
  if less: ⟨i j⟩ then i
  else j
```

can be abbreviated by

```
min: ⟨i j⟩ ≡
  less: ⟨i j⟩ i
  j.
```

The tests are evaluated in sequence until one succeeds; the value immediately following that test is the value of the function. If no test succeeds then the value of the function is the value of the last expression in the sequence if the sequence is of odd length (the else part), or rarely the empty list if the sequence is of even length.

As an example we present the definition of the function allrember which *rem*oves *all* mem*bers* equal to its first argument from the list which is its second argument.

```
allrember: ⟨e l⟩ ≡
  if null: ⟨l⟩ then ⟨ ⟩
  elseif same: ⟨first: ⟨l⟩ e⟩
        then allrember: ⟨e rest: ⟨l⟩⟩
  else cons: ⟨first: ⟨l⟩
                allrember: ⟨e rest: ⟨l⟩⟩⟩.
```

It is also possible to define functions which take an arbitrary number of arguments in the same manner. An example is the function concat which returns a list which is the concatenation of all its arguments (each of which is a list). An auxiliary function, append, is required which concatenates just two lists.

```
concat: ls ≡
  if null: ⟨ls⟩ then ⟨ ⟩
  elseif null: ⟨rest: ⟨ls⟩⟩
        then first: ⟨ls⟩
  else append: ⟨first: ⟨ls⟩
                  concat: rest: ⟨ls⟩⟩;
append: ⟨la lb⟩ ≡
  if null: ⟨la⟩ then lb
  else cons: ⟨first: ⟨la⟩
                append: ⟨rest: ⟨la⟩ lb⟩⟩.
```

Integers may be used as functions: as a function the integer *i* simply returns its *i*th argument. One use of this notation provides for array subscripting: if *c* is bound to a list of lists (a matrix) then 3: 5: *c* evaluates to the third item in the fifth list (or the entry in the third column of the fifth

row). The integer 1 may also be used as an identity function, often with the "invisible argument marker" symbol #.

The symbol # evaluates to a token which is ignored as a parameter to a function. Its evaluation is therefore useless except as an eventual argument to some function; in that role it acts much like the numeral zero: as a place-holder in argument structures with no ultimate meaning itself. For example, if d is bound to the evaluation of ⟨# # 9 # 15 # #⟩ then $1{:}d$ evaluates to 9, $3{:}d$ diverges since there is no third item in d taken as a parameter list. A list like d is often used in conjunction with functional combination (below).

Functional Combination

Functional combination is described elsewhere in some detail [6] and [7]. It provides the framework which allows one recurrence to accumulate results in the same way that a single iterative traversal of data may yield several summary statistics. We describe its syntax and semantics formally here. The hallmark of functional combination is the occurrence of a list in the function position. In first-order languages (where forms cannot evaluate to functions) this can only happen if an explicit list (within brackets) appears where a function is expected:

$$\langle f_1 \;\; f_2 \;\; \cdots \;\; f_{m_0}\rangle : \langle \rho_1 \;\; \rho_2 \;\; \cdots \;\; \rho_n\rangle.$$

The list immediately to the left of the colon is called a *combinator* and is not evaluated. Instead each f_j is presumed to be a legitimate function; either it has a definition as a function or it too is a combinator. Any f_j must require at most n arguments; its arguments are extracted from the structure of the arguments ρ_i to the combinator; each one is presumed to be a list.

The semantics of functional combination depends on the length of the arguments and the combinator itself. Let m_i be the length of ρ_i, the ith row. Let

$$m = \min_{0 \le i \le n} m_i.$$

The result of evaluating the form with a combinator as its function is a list of length m. The jth element in that list is the result of

$$f_j : \langle j' : \rho_1 \;\; j' : \rho_2 \;\; \cdots \;\; j' : \rho_n\rangle.$$

(The integer function j' is the same as the function j except that a token evaluation of # is counted in selecting the result. If the result of applying j' is an instance of #, it is passed as a parameter to f_j, which ignores it.)

In full blown form we have

$$\begin{aligned}\langle f_1 \;\; f_2 \;\; \cdots \;\; f_{m_0}\rangle : \langle \rho_1 \;\; \rho_2 \;\; \cdots \;\; \rho_n\rangle \\ = \langle f_1 : \langle 1' : \rho_1 \;\; 1' : \rho_2 \;\; \cdots \;\; 1' : \rho_n\rangle \\ f_2 : \langle 2' : \rho_1 \;\; 2' : \rho_2 \;\; \cdots \;\; 2' : \rho_n\rangle \\ \vdots \qquad \vdots \qquad\quad \vdots \qquad\qquad\quad \vdots \\ f_m : \langle m' : \rho_1 \;\; m' : \rho_2 \;\; \cdots \;\; m' : \rho_n\rangle\rangle.\end{aligned}$$

An elegant interpretation of the evaluation of such a form arises from viewing the result of evaluating each ρ_i as the ith row of a matrix whose columns are then referred to as γ_j for $1 \le j \le m$. The result of evaluating the entire form is that of

$$\langle f_1 : \gamma_1 \;\; f_2 : \gamma_2 \;\; \cdots \;\; f_m : \gamma_m\rangle.$$

Thus, the evaluation procedure can be described as an evaluation of arguments in row-major order with parameters passed to functions in column-major order. The derivation of m as a minimum implies a "guillotine rule" which causes a "jagged" matrix of arguments to be truncated at the narrowest width. In the case that $m = 0$ the result of the evaluation is the empty list. As an immediate result the list ⟨ ⟩ is defined as a constant function: ⟨ ⟩$:l$ evaluates to the empty list regardless of the binding of l.

In order to facilitate the matrix interpretation of functional combination its invocation will appear with the arguments on separate lines and vertically aligned to suggest the columnar relationship. Furthermore, names of functions which return results of fixed length will be hyphenated to suggest the meaning of each component of the answer. For example:

```
⟨sum product quotient difference⟩:⟨
  ⟨ 0     1        63        19    ⟩
  ⟨ 1     3        #         11    ⟩
  ⟨ 0     3        9         #     ⟩⟩
```

evaluates to (1 9 7 8).

A more interesting example illustrates the power of functional combination as related to recursive programming. The function lt-eq-gt takes a list of numbers and a numeric value as parameters and returns three results corresponding to the three components of the partition of the list by that value: those less than, those equal to, and those greater than it. The construction of the partition is accomplished by a single linear recursion over the list. Since operations like this are common in programming (for example, it is the key step in the Quicksort algorithm [14]) it is important that they be expressible in a form analogous to the simple loop available to iterative programmers.

The following example uses functional combination three times in essentially the same way: the pattern of invocation is ⟨···⟩:⟨···⟩ which suits the row/column description given before. An invocation may also appear as ⟨···⟩$:l$ where l is bound to a matrix which will be decomposed to extract parameters in the manner described above. It is also possible to write something of the form ⟨···⟩$:f:l$ which indicates that the matrix will be the result of invoking a second function f on l.

```
lt-eq-gt:⟨l v⟩     ≡
  if null:⟨l⟩ then ⟨⟨ ⟩⟨ ⟩⟨ ⟩⟩
  elseif less:⟨first:⟨l⟩ v⟩
       then cons   1  1      ⟩:⟨
            ⟨first:⟨l⟩ # #⟩
            lt-eq-gt:⟨rest:⟨l⟩ v⟩⟩
  elseif greater:⟨first:⟨l⟩ v⟩
       then ⟨⟨1   1   cons        ⟩:⟨
            ⟨# # first:⟨l⟩⟩
            lt-eq-gt:⟨rest:⟨l⟩ v⟩⟩
  else ⟨1   cons        1⟩:⟨
       ⟨# first:⟨l⟩ #⟩
       lt-eq-gt:⟨rest:⟨l⟩ v⟩⟩.
```

Another application of functional combination involves the invocation of the function being recursively defined with the combinator. We present an example in which the defined function appears twice, resulting in two recursive invocations. In a deep recursion the invocation pattern generates a binary tree: at the nth level the results are determined by the results of 2^n functional combinations which dispatch 2^{n+1} recursive calls. That tree structure is no accident since the example is concerned with searching binary trees [18] (those whose in order [19] traversal visits the nodes in order of their keys). Let l be an unsorted list of perhaps duplicated keys. We present a function, quickbatch, which probes *tree* to extract any information for every key in l and returns a list of the associations for those keys which had information planted in *tree*. The list will be returned in ascending order of keys; and the search will be batched [23], so that every subtree is visited at most once.

Define a binary tree to be ( ) or a list of three items: (left information right). *Information* represents the data stored at the root of the tree whose subtrees are *left* and *right*, respectively. In this case information is an association of *key* and data. The invocation ⟨key tree⟩ extracts the key from the root of the nonnull tree; the definition requires that this key be greater than every key in the *left* subtree and less than any in the *right* subtree.

```
quickbatch: ⟨l tree⟩ ≡
  if null: ⟨l⟩ then l
  elseif null: ⟨tree⟩ then ⟨ ⟩
  else concat:
       ⟨quickbatch hit quickbatch⟩: ⟨
       lt-eq-gt: ⟨l key: ⟨tree⟩⟩
            tree                   ⟩:
  hit: ⟨l info⟩ ≡
    if null: ⟨l⟩ then ⟨ ⟩
    else ⟨info⟩.
```

The last line of quickbatch deserves some explanation. The result of the use of functional combination is three lists of associations on keys which are to be concatenated. The first and third are derived from recursive calls on the left and right subtrees of the nonnull tree. The middle list is empty unless the key found at the root of the tree happened to be mentioned once or more in the target list of the search. Finally, the sorting of the answer list is carried out by an implicit Quicksort at each node in the search tree. The function lt-eq-gt partitions at key: tree the target list carried in an unordered batch to *tree*. For example, if *tree* is as shown in Fig. 1, then quickbatch: ⟨⟨9 2 3 6 8 7 3⟩ *tree*⟩ evaluates to ((2 ant)(3 boa)(8 eel)(9 dor)).

STARS

The next language feature is called "star" because of its syntax, reminiscent of the Kleene star. If z evaluates to A then the list $\langle z^*\rangle$ evaluates to the list $(A^*) = (A\ A\ A\ A \cdots)$, which has the semantics of a list of an infinite number of A's, although it may be represented in finite space and printed in finite (star) notation. Similarly, ⟨0*⟩ evaluates to

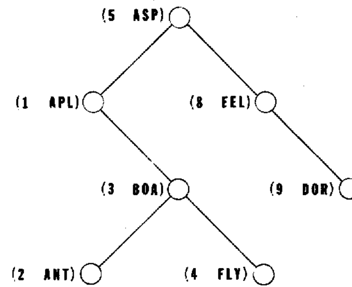

Fig. 1.

an infinite list of zeros (the zero vector) which, fortunately, may be printed as (0*); $\langle x^*\rangle$, under the binding of x as (2 4 6 8), evaluates to a matrix with an infinite number of rows and only four columns which may be printed: ((2 4 6 8)*).

The star notation may be applied in constructing combinators if all elements are identical. For instance, in order to add one to every element of a vector x one can write

```
⟨sum*⟩: ⟨
⟨ 1* ⟩
  x   ⟩
```

which evaluates to (3 5 7 9) under our binding for x. The definition of functional combination above still applies under the convention that the values m_i can be infinity for starred rows. In the previous example $m_0 = \infty = m_1$ and $m_2 = 4$ so $m = 4$. Of course, if all $m_i = \infty$, then $m = \infty$, as established by the convention

$$\langle f^*\rangle : \langle \langle \alpha_1^*\rangle\ \langle \alpha_2^*\rangle \ \vdots\ \langle \alpha_n^*\rangle\rangle = \langle f : \langle \alpha_1\ \ \alpha_2\ \cdots\ \alpha_n\rangle^*\rangle.$$

The star notation may be applied only to the suffix of a list whose prefix is explicitly expressed: ⟨cons cons sum*⟩ is a legal combinator and ⟨2 3 4 5*⟩ evaluates to (2 3 4 5 5 5 5 ⋯).

Starred structures are most useful in the context of functional combination. Starred functions are "spread" (or mapped [21]) across all available columns of the argument matrix; starred arguments are shared by all columns. As an example of the impact of stars we present Gaussian matrix multiplication, leaving the definition of transpose to the reader.

```
dotproduct: ⟨v1 v2⟩ ≡ sum: ⟨product*⟩: ⟨
                                v1
                                v2        ⟩;
row: ⟨vec transp⟩ ≡ ⟨dotproduct*⟩: ⟨
                      ⟨vec*⟩
                      transp    ⟩;
mtxmpy: ⟨m1 m2⟩ ≡ ⟨row*         ⟩: ⟨
                    m1
                    ⟨transpose: ⟨m2⟩*⟩⟩.
```

THE ROLE OF SUSPENDING CONS

The function cons is representative of an entire class of functions which build structures by filling in the values of fields within nodes. Syntactically it also serves as a space allocator although that characteristic plays a lesser role in the following discussion. We have proposed a new semantics for cons and its extractor functions first and rest which avoids the construction of those portions of structures that are never accessed after their creation. The results apply to any operation which assigns a value to a field, provided that it is possible to preserve a record of all relevant bindings. This criterion is difficult to meet in a system in which users can change assigned values, but it is easily satisfied under a regime of applicative programming in which the user can only create and implicitly release such bindings [15].

Using the function cons as a paradigm of structure-creating functions, we briefly explain its semantics. When cons is invoked by the user, the value returned is a pointer to a newly built structure. Rather than evaluate the arguments to cons and create the complete structure, we create a structure consisting of two *suspensions.* A suspension consists of a reference to the form whose evaluation was deferred and a reference to the environment of variable bindings in which the suspension was originally created. These two structures must remain intact for the life of the suspension. The reference to the form is a pointer to a piece of program, so the space it occupies usually represents no great overhead. Environments present more of a problem, since we are accustomed to viewing them only as temporary structures. Moreover, use of destructive assignment operations generally requires recreation of the entire environment in order to assure the integrity of references to the environment as it existed before the assignment. Destructive assignments, if not well controlled, become costly; it is fortunate that they do not exist in our source language.

When either of the functions first or rest is invoked, the following events occur. A designated field of the argument is checked to determine if it contains a suspension (suspensions are flagged and easily distinguished); if not, then its content is returned. If a suspension is present, then the evaluator is invoked upon the designated form within the preserved environment. The result is stored back in the designated field in place of the suspension (for next time), and the value is returned as a final result. These events constitute *coercion* of the suspension. The two functions, first and rest, therefore act as probes into the data structure, with possible effects of a predictable and benign sort, rather than as simple extractor functions.

As a result of suspending, evaluations are delayed as long as possible. Ultimately all evaluations take place as a result of the demands of the driver of the output device which tries to move the contents of its list to the external device. As it traverses the list it is outputting, it invokes first and rest, causing top-level evaluation, which in turn results in the creation and inspection of more structure, indirectly forcing all of the necessary evaluations. Regardless of the intentions of the programmer, the only structures which are actually built are those which are essential to deciding what information is to be output. Assuming that conditional expressions have their usual interpretation [25, p. 335] of being strict in their first parameter [8, p. 261], least fixed-point semantics for the language results [8, p. 270].

A fortunate side effect of suspending the creation of data structures is the ability to deal with infinite structures. Consider the list defined (but never completely constructed) by the invocation of terms: $\langle 0 \rangle$ where

$$\text{terms}:\langle n\rangle \equiv \text{cons}:\langle \text{recip}:\langle \text{square}:\langle n\rangle\rangle \;\; \text{terms}:\langle \text{sum}:\langle 1\ n\rangle\rangle\rangle.$$

That list, the reciprocals of the squares of all the positive integers, might be familiar since its sum, excluding the first term, converges to $\pi^2/6$. Suppose that z were bound to the result of terms: $\langle 0 \rangle$; in fact, because of the suspending cons, z is initially bound only to a "promise" of this result. As long as $1:z$ is not computed (since it diverges on division by zero) and as long as a complete traversal of the structure is not invoked, the infiniteness of z poses no problem. An access to $6:z$, if essential to the output device, would find the answer 0.04 even though that number had not been present before that access; it would have been computed had it been of interest earlier. (This use of cons is similar to Landin's prefixs developed by Burge [3], but it differs precisely in that the rest of the list z may be accessed without computing the divergent first element.) There are other implications of cons [11] for infinite structures [17].

The same techniques used for cons may be applied to any record creating (field assignment) function within the system. We have proposed an interpreter [8] in which all field assignments are suspended. This has a great impact; in particular the construction of environments may be suspended. This means that no argument will be evaluated unless the corresponding formal parameter has been accessed by some operation critical to the execution of the program (i.e., critical to the creation of output). This effects the call-by-need argument-passing protocol [26], the call-by-delayed value [25], and lazy evaluation scheme [12].

Another effect is on the semantics of functional combination. The result of an application of functional combination is a list which, not surprisingly, is conventionally built with cons. If cons suspends evaluation then only those items in that list which are accessed are ever created. For instance, the result of an invocation of quickbatch is a list. That list, if not trivial, is the result of an invocation of concat which uses cons. Later arguments to concat need not all be computed at once (or even at all if only a part of the result were ever needed for printing). The argument list for concat is the result of functional combination and thus, as we suggest here and demonstrate elsewhere [6], [7] *need* not be computed all at once. Instead of computing the complete answer, only that computation path essential to the answer is pursued. Intermediate environments are preserved in case any suspensions are coerced later. Recursive calls on the left and

the right subtrees often need not both be evaluated. For instance, only five recursive calls on quickbatch are required to determine the first information-pair, (2 ant), in the example above which requires fourteen recursive calls (plus the outermost call) in order to ascertain the final answer.

Opportunities for Parallelism

With the language defined, we now turn to the *opportunities* for parallelism provided in the language. We do not explicitly require these parallelisms to be performed, nor do we require that the programmer be aware that they even *may* occur. Programs are easily written with these control structures with the semantics described in the previous section, which do not depend on concurrencies. It is significant that some of the semantics of the language allow for improvements in parallel interpretation of programs written in a very popular language differing only syntactically from ours [21]. It is the role of a compiler to detect the opportunities for parallelism in its pass over the program before run time and to alter the code to be interpreted in order to provide for the parallelism allowed by the target hardware. The responsibilities for synchronization are therefore the concern of the compiler so the programmer need not worry about issues of "structured multiprogramming" [2].

At the same time that we say that the compiler should detect parallelism for run time, we should point out how the source language helps the compiler in this task by allowing simple program structures. Most notably, the language does not have destructive assignment statements; it is free of side effects. All variable/value bindings are established as parameter/argument bindings in function linkages, and they are therefore not subject to change during their lifetime. An obvious (but not new [27]) opportunity for parallelism is collateral argument evaluation, establishing these bindings simultaneously since they are defined independently of one another. These bindings are abandoned after all computations under the environment of the function invocation have been completed, but until then they remain intact. This integrity of environments, essential to the suspending cons, also alleviates the concurrency problem, since no conflict arises because of a reader accessing a value as a writer alters it [4].

The feature of suspending cons, itself, provides opportunity for massive parallelism. A system implemented with only the user's invocations of cons suspended, or with those and all the system structures suspended, may have hundreds of suspensions pending on the system during the course of computation. In a single processor system all (but one) of these would await probing by the system functions, first and rest, before their coercion would be initiated. If the run-time environment were enriched with idle processors, then *any* of these suspensions could be coerced simultaneously without delaying the progress of the critical evaluation (the single one active on a single processor). Let us designate that distinguished evaluation as the *colonel* and any other processors available will be called *sergeants.*

The parallel evaluation strategy is to keep the colonel working on the same critical process which would occupy a single processor and to allocate the sergeants to suspensions which are "near" the colonel process. Since evaluation of suspensions usually converges to nodes containing new suspensions rather quickly, sergeants tend to finish tasks rapidly after which they are reassigned to new ones "closer" to the moving colonel. (It is possible that a sergeant could fall into a divergent evaluation and therefore be lost to the system until the suspension it was evaluating becomes irrelevant.) The colonel behaves exactly as a single processor would, except that from time to time it accesses what would have been a suspension and instead finds the result already provided by a sergeant who had passed through earlier. The definition of the "near" metric should be chosen to maximize the likelihood of this fortunate event. The sergeants scurry about the system following the colonel doing their best to satisfy his anticipated needs. Some of their effort may be wasted since not all handiwork of sergeants need be accessed by the colonel. Yet the time to compute the final result is no more than the time using a single evaluator since parallelism has been provided at essentially no overhead. There is no cost due to interprocessor communication, and little interprocessor conflict depending on memory architecture and access pattern. Some additional cost may arise from the enforcement of the "near" metric; but this requires overhead only as a sergeant process is initiated—not while it is running.

Even though a processor has been led down the garden-path (diverges) [8], there is still an opportunity for recovery, if the value it is supposedly computing is discovered to have become unnecessary to the system. This, in fact, is rather easily accomplished because processor allocation is so closely tied to the data structure. The same mechanism which determines that a node has become useless and is to be returned to available space need only stop execution of any process (some wayward sergeant) which is operating on a suspension referenced from that node. Since all space allocated by the colonel for its computation will be returned after the result has been provided, it follows that all sergeants will be recovered as well by that time. Therefore, if the colonel's computation converges it is not possible to lose a sergeant; all space and processors will be restored to the system.

Functional combination offers two sorts of parallelism. The first is exemplified by the code for lt-eq-gt. In the definition for this function the recursion is linear down the list parameter, but at each recursion step each of the three developing results must be handled. Clearly the three pieces of the final result can be handled by three concurrent processes. So a simple but bounded parallelism is provided depending on the size (m in the definition above) of the result when all elements of the combinator are defined independently of the function definition in which the combinator appears.

Another kind of parallelism results if that function itself appears in the combinator. The coding of the function quickbatch is an example of this. If m processors are allocated for computing the result of a combinator and a combinator has occurrences of the function being defined as some f_j, then a process tree can result with processors active only at the leaves. The reason a tree is produced is that a

single processor evaluating a recursive function might encounter an instance of functional combination and become dormant while the m processes from that instance compute. If some of those processes are recursive invocations, then each of those processes may become dormant in the same way. If all processes terminate then the invocation tree is of finite depth with degree m at any node, with dormant processes at all nonleaves, and with active processes only at the leaves. If a combinator has more than one recursive call in such a scheme then a very "bushy" process tree can result. For example, the quickbatch function of the Quicksort Algorithm can be implemented so that every recursion requires a new processor. At the nth level 2^n processors may be required. The processors are all evaluating the same function definition under disjoint (and static) environments, however, so that *lock-step* evaluation is entirely appropriate.

These semantics require very little *inter*processor protocol. Upon interpretation of functional combination the active process goes dormant and spawns m new processes. Each of these processes is independent and need not initiate communication with any other user process except to report its result. As it reports its result a process dies but its dormant parent is jarred; we call this process *stinging.* A stung parent becomes active when it is stung with the (chronologically) last result. Therefore, the only run-time processor synchronization involves process creation and stinging. (Environments are static!) This is no more complicated than what is required for collateral argument evaluation.

The star notation used on an argument to functional combination merely denotes that the argument is to be shared by all m processors. When the combinator itself is a starred structure then the combinator is homogeneous and a different sort of concurrency may be used for interpreting the function. This use of combinators is most similar to mapping functions in [21] and their generalization in [24]. An example is the code for dotproduct above in which all additions may take place concurrently. Due to the expression of the combinator with star, the compiler can easily detect that the same operation will be performed on all objects in the data structures which are arguments to the combinator. Then the evaluation may proceed using *pipelining* across the n arguments to the starred combinator.

Similarly, the starred notation used within the combinator itself denotes that the code for the function is to be used by each of the m processors. Under parallel interpretation this kind of functional combination has the semantics of shared pure code. For instance, the algorithm for mtxmpy specifies that the code for the function row is to be shared by all processors used in interpreting its functional combination, i.e., by up to M processors in an M by N matrix times an N by P matrix problem. Also, row specifies that the code for dotproduct can be shared by the up to N processors used for its functional combination, where each of these is starred combinator distributing the code for the primitive instruction product across P processors. Thus up to $N \times M \times P$ multiplications might be performed simultaneously by processors interpreting the shared code in parallel.

Conclusions

Functional combination allows the use of known forms of controlled parallelism, whereas the suspending cons will allow masses of sergeant processors to be occupied on heuristically useful computation. The former facility fits existing hardware which now requires specific higher level languages and specially trained programmers in order to occupy the processors productively. The latter approach offers a hope for occupying a machine with arbitrarily large numbers of processors whose temporal configuration cannot be known to the programmer.

This ability of our semantics to use a system with massive parallelism (thousands of processors) is very important for future hardware design. Such systems will not be built unless there is a way to program them, even though the current cost of processors suggests that they will be technically possible. With communication cost high and processor cost negligible, pressure will build for a massive computation on data while they remain within storage directly accessible to any processor. Not only do our semantics admit such massive (albeit heuristic) parallelism, but also they achieve these results on a well-known language, pure Lisp, imposing these semantics on programs extant fifteen years ago.

Taken together these approaches to programming in purely applicative source code provide the programmer with higher level tools for expressing algorithms so that the compiler can recognize and compile parallel code.

Acknowledgment

The authors wish to thank C. Brown who helped refine this paper. Thanks are also due to S. Smoliar who pointed out the implications of functional combination for sharing code.

References

[1] J. Backus, "Programming language semantics and closed applicative languages," in *Proc. ACM Symp. Principles of Programming Languages*, pp. 71-86, 1973.
[2] P. B. Hansen, "Concurrent programming concepts," *Computing Surv.*, vol. 5, no. 4, pp. 223-245, 1973.
[3] W. H. Burge, *Recursive Programming Techniques.* Reading, MA: Addison-Wesley, 1975.
[4] P. J. Courtois, F. Heymans, and D. L. Parnas, "Concurrent control with 'readers' and 'writers'," *Commun. Assoc. Comput. Mach.*, vol. 14, no. 10, pp. 667-668, 1971.
[5] D. P. Friedman and D. S. Wise, "Unwinding stylized recursions into iterations," Comput. Sci. Dep., Indiana University, Bloomington, IN, Tech. Rep. 19, 1975.
[6] ——, "Functional combination," *Computer Languages*, to be published.
[7] ——, "An environment for multiple-valued recursive procedures," in *Programmation.* Paris, France: Dunod Informatique, 1976.
[8] ——, "*CONS* should not evaluate its arguments," in *Automata, Languages and Programming*, S. Michaelson and R. Milner, Eds. Edinburgh, Scotland: Edinburgh Univ. Press, 1976, pp. 257-284.
[9] ——, "Output driven interpretation of recursive programs, or writing creates and destroys data structures," *Inform. Processing Lett.*, vol. 5, no. 6, pp. 155-160, 1976.
[10] ——, "Aspects of applicative programming for file systems," in *Proc. ACM Conf. Language Design for Reliable Software, SIGPLAN Notices*, vol. 12, pp. 41-55, 1977.
[11] D. P. Friedman, D. S. Wise, and M. Wand, "Recursive programming through table look-up," in *Proc. ACM Symp. Symbolic and Algebraic Computation*, pp. 85-89, 1976.
[12] P. Henderson and J. Morris, Jr., "A lazy evaluator," in *Proc. 3rd ACM Symp. Principles of Programming Languages*, pp. 95-103, 1976.

[13] C. E. Hewitt and B. Smith, "Towards a programming apprentice," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 26–45, Jan. 1975.
[14] C. A. R. Hoare, "Quicksort," *Computer J.*, vol. 5, no. 1, pp. 10–15, 1962.
[15] ——, "Towards a theory of parallel programming," in *Operating Systems Techniques*, C. A. R. Hoare and R. H. Perrott, Eds. London: Academic Press, 1972, pp. 61–71.
[16] S. D. Johnson, "An interpretive language based on suspended computation," M.S. Thesis, Dep. Comput. Sci., Indiana Univ., Bloomington, IN, 1977.
[17] G. Kahn and D. MacQueen, "Coroutines and networks of parallel processes," in *Information Processing 77*, B. Gilchrist, Ed. Amsterdam, The Netherlands: North-Holland, 1977, pp. 993–998.
[18] D. E. Knuth, *Sorting and Searching*. Reading, MA: Addison-Wesley, 1973.
[19] ——, *Fundamental Algorithms* (2nd ed.). Reading, MA: Addison-Wesley, 1975.
[20] P. J. Landin, "A correspondence between ALGOL 60 and Church's lambda notation—Part I," *Commun. Assoc. Comput. Mach.*, vol. 8, no. 2, pp. 89–101, 1965.
[21] J. McCarthy, P. W. Abrahams, D. J. Edwards, T. P. Hart, and M. I. Levin, *LISP 1.5 Programmer's Manual*. Cambridge, MA: M.I.T. Press, 1962.
[22] S. S. Patil, "An abstract parallel-processing system," M.S. Thesis, Dep. Elec. Eng., M.I.T., Cambridge, MA, 1967.
[23] B. Shneiderman and V. Goodman, "Searching of sequential and tree structured files," *ACM Trans. Database Syst.*, vol. 1, no. 8, 1976.
[24] G. Tesler and H. J. Enea, "A language design for concurrent processes," in *Proc. Spring Joint Comput. Conf.* Washington, DC: Thompson, 1968, pp. 403–408.
[25] J. Vuillemin, "Correct and optimal implementation of recursion in a simple programming language," *J. Comput. Sys. Sci.*, vol. 9, no. 3, pp. 332–354, 1974.
[26] C. Wadsworth, "Semantics and pragmatics of lambda-calculus," Ph.D. dissertation, Oxford Univ., Oxford, England, 1971.
[27] A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, and R. G. Fisker, "Revised report on the algorithmic language ALGOL 68," *Acta Inform.*, vol. 5, no. 1-3, pp. 1–236, 1975.

Daniel P. Friedman was born in Middletown, CT, on August 16, 1944. He received the B.S. degree in mathematics from the University of Houston, Houston, TX, in 1967, and the M.A. and Ph.D. degrees in computer science from the University of Texas, Austin, in 1969 and 1973, respectively.

From 1968 to 1969 he was an instructor in the Department of Computer Science, University of Houston, and from 1971 to 1973 he was an Assistant Professor at the Lyndon Baines Johnson School of Public Affairs. In 1973 he became an Assistant Professor in the Department of Computer Science, Indiana University, Bloomington. His research interests include robotics, semantics, graph processing, and applicative programming languages.

David S. Wise (M'71) was born in Findlay, OH, on August 10, 1945. He received the B.S. degree in mathematics from Carnegie Institute of Technology, Pittsburgh, PA, in 1967, and the M.S. and Ph.D. degrees in computer sciences from the University of Wisconsin, Madison, in 1969 and 1971, respectively.

From 1971 to 1972 he was a lecturer in the Computer Science Department at the University of Edinburgh. In 1972 he joined Indiana University, Bloomington, where he is now Associate Professor. His research has been in the fields of formal grammars and programming languages. His presentation of this paper at the 1976 International Conference on Parallel Processing, Waldenwoods, MI was voted best of the conference.

Dr. Wise is a member of the Association for Computing Machinery, Omicron Delta Kappa, and Tau Beta Pi.