Preliminary Approach to
Applicative Programming for Systems*

Daniel P. Friedman
David S. Wise

Computer Science Department
Indiana University
Bloomington, Indiana 47401

TECHNICAL REPORT NO. 78
PRELIMINARY APPROACH TO
APPLICATIVE PROGRAMMING FOR SYSTEMS

DANIEL P. FRIEDMAN
DAVID S. WISE
NOVEMBER, 1978

---

*This note is excised from a grant proposal submitted in May, 1977, to the National Science Foundation. The authors' ideas have been refined since, and the definitions given here may be obsolete. Research reported herein was supported (in part) by the National Science Foundation under grants numbered DCR75-06678 and MCS75-08145.

Abstract

Applicative languages have attracted considerable attention because of formal properties relating to important theoretical problems of program support. For instance, program verification and program transformation techniques are the most developed for such languages. This research is intended to develop the power of such expressive styles so that they can conveniently handle "systems" problems (e.g. operating and data base systems) which involve real time and concurrency, as well as asynchronous parallel processing, through variations of "suspended" data structures. A significant goal is a natural style in an applicative language which is an attractive tool for difficult problems of this sort.

Background

The development of programming models can be separated into two categories: iterative and applicative. Both models of computation have roots in mathematical logic: Turing machines for the former and recursive functions [39] for the latter, but hardware design and then most language design followed the iterative model. Languages like FORTRAN, ALGOL, and PL/1 exemplify the iterative philosophy. Language models like the lambda-calculus [6], and combinators [8] as developed by the RED languages [1], pure LISP [32], ISWIM [28,4], Plasma [20] and Data Flow [9] exemplify the applicative philosophy.

A fruitful and promising area of formal research has evolved around the applicative model, yet almost all programming is done using the iterative model. Recent results which rely on an applicative model of programming are the fixed-point semantics of Scott [40] and deBakker [2], subgoal induction by Morris and Wegbreit [34], program transformation by Burstall and Darlington [5], cost analysis by Wegbreit [48], program generation by Summers [41], program proving by Boyer and Moore [3], and work related to business data processing by Leavenworth [29]. More relevant to our approach are recent observations on argument evaluation strategies and their effect on the course of computation. This issue is developed by Vuillemin [44], Wadsworth [45] and refined by Henderson and Morris [19], Friedman and Wise [13], Kahn [23] and Kahn and MacQueen [24].

Aspects of Applicative Programming

The key question addressed by this research is whether it is feasible (under practical standards) to perceive all programming problems through applicative programming. By applicative programming we mean a style in which the only control structure is a function invocation, the only bindings are of arguments to formal parameters, and the only effects are function results. Others have characterized it with referential transparency [36] or by its absence of side effects (like the assignment statement) [9]. We adopt a particular syntax for the discussions here. It is at once similar to RED [1], and ISWIM [28]. Specifically, the structure of functions and arguments is from RED, but the use of formal parameters is from ISWIM.

Functionally expressed programs do not necessarily perform using the stacking mechanism [10] which is the common implementation mechanism. Tail-recursion [42], continuation passing [37], and the program transformations of Burstall and Darlington [5] each yield some efficiency over stack implementations. While we are interested in furthering compilation techniques of this nature, that issue is not fundamental. We perceive the critical issue as one of style: what tools will the programmer be given and what directions will he have for their use. Proper guidance will encourage him to express himself easily while creating programs which yield to program transformations, both known and yet undiscovered.

The remainder of this report confronts these two issues, tools and style. Compilers are an eventual goal, but it is critical that all implementation possibilities be reflected in language design insofar as the compiler writer is concerned. We choose to describe these possibilities from the perspective of the user, who is given only tools and style, in order to sharpen the contrast between what is written as programs and what actually happens when that code is interpreted. (We are not avoiding implementation; we avoid a specific implementation.)

The important question in evaluating this user perspective is the power of his applicative expression. In a recent paper [17] we demonstrated that an interactive editor could be programmed under our semantics of computation. Here we present sample problems from operating and database systems: an interrupt handler and an n-way merge of sorted files. While these programs are only representative, they demonstrate the convenience of user expression and the depth of program transformation admitted even by direct interpretation of untransformed code. We believe that hard problems are best solved using applicative style and more importantly, that efficient implementations on hardware can follow automatically [15] from such code. In most instances applicative code should be sufficient to yield an efficient solution without reprogramming in a sequential (iterative) language. Before proceeding to problems raised by the application of our computational model to state-of-the-art programming problems we shall briefly review our applicative language.

Notations and Definitions

The computational model is composed of atoms, functions, denoted f, and structures collectively referred to as elements denoted e. The structures are further partitioned into sequences denoted s, multisets denoted m, and hybrids denoted h.

Special Atoms

The special atoms are TRUE, FALSE and UNDEFINED.

Sequences

A sequence s of elements $e_1, e_2, \ldots, e_k$ is denoted by

$$<e_1 e_2 \ldots e_k>.$$

An infinite sequence $<e_1\ e_2 \ldots e_k{*}>$ is the same as

$$<e_1 e_2 \ldots e_k e_k e_k \ldots>.$$

Multisets

A multiset m of elements $e_1, e_2, \ldots, e_k$ is denoted [25] by

$$\{e_1 e_2 \ldots e_k\}.$$

An infinite multiset $\{e_1 e_2 \ldots e_k{*}\}$ is the same as

$$\{e_1 e_2 \ldots e_k e_k e_k \ldots\}.$$

A multiset is unordered; a sequence is totally ordered.

Hybrids

A hybrid h of elements $e_1 e_2, \ldots, e_i, e_{i+1}, \ldots, e_k$ is denoted by $<e_1 e_2 \ldots e_i | e_{i+1} \ldots e_k\}$. The first i elements are treated like a sequence and the remaining elements are treated like a multiset, with the order on the sequence extended so all sequence elements precede multiset elements.

The following identities hold:

when $i = k$, $<e_1 e_2 \ldots e_k | \} \equiv <e_1 e_2 \ldots e_k>$;

when $i = 0$, $< | e_1 e_2 \ldots e_k \} \equiv \{e_1 e_2 \ldots e_k\}$; and

when $k = 0$, $<|\}$ is the empty hybrid, which is the same as $\{\}$ and $<>$; we denote this empty structure by $\square$.

Application

An application requires two elements, a function f and a structure h and is denoted by f:h. For example quotient:<12 3> is 4.

Elementary functions

Let $s = <e_1 e_2 \ldots>$ and $m = \{e_1 e_2 \ldots\}$ then

first:s = e_1 or UNDEFINED if $s = \square$;[†]

rest:s = $<e_2 \ldots>$ or UNDEFINED if $s = \square$;[†] and

cons:<e s> = $<e\ e_1 e_2 \ldots>$.

On evaluation of either first:m or rest:m where m is a multiset, m becomes a hybrid structure $<e_c | e_1 e_2 \ldots e_{c-1} e_{c+1} \ldots\}$. The arguments specifying elements of m are evaluated asynchronously 'in parallel'. The 'first' element to converge, e_c, becomes the permanent *first* element of m, now a hybrid. Then

first:m = e_c or UNDEFINED if $m = \square$;

rest:m = $\{e_1 e_2 \ldots e_{c-1}, e_{c+1} \ldots\}$ or UNDEFINED if $m = \square$.

[†] The LISP function *car* and *cdr* can be defined in terms of *first* and *rest*. Let ℓ be a parameter list of one element (itself a list)

then car:ℓ ≡ first:first:ℓ and
cdr:ℓ ≡ rest:first:ℓ.

cons:<e m> = <e|e_1e_2...};

mons:<e m> = {e e_1e_2...};

empty?:h = is TRUE if h is □ or FALSE otherwise;

atom?:e = is TRUE if e is an atom or FALSE otherwise; and

same?:h = TRUE if h is composed only of atoms and the atoms are the same (under some equality relation); FALSE if some differ; or UNDEFINED otherwise.

Functional Forms

Composition

Composition of functions associates to the right. We define integers as functions which return the specified item from a sequence (forming a multiset into a hybrid to determine the prefix sequence). $\text{First:}\underbrace{\text{rest:rest:}\ldots\text{:rest}}_{\text{i-1 times}}\text{:h}$ is synonomous with i:h.

Having shown that sequences and multisets are special cases of hybrids we redefine the primitives. Let h = <$e_1e_2 \ldots e_i$|$e_{i+1} \ldots e_k$}

first:h = if i>0 then e_1

else if i = 0 then e_c with h redefined as

<e_c|$e_1e_2 \ldots e_{c-1}, e_{c+1} \ldots e_k$}

else UNDEFINED;

rest:h = if $i>o$ then $<e_2 \ldots e_c | e_{c+1} \ldots e_k\}$

else if $i = 0$ then $<| e_1 \ldots e_{c-1}\ e_{c+1} \ldots e_k\}$ with h redefined as

$$<e_c | e_1 e_2 \ldots e_{c-1}\ e_{c+1} \ldots e_k\}$$

else UNDEFINED;

cons:$<e\ h> = <e\ e_1 e_2 \ldots e_i | e_{i+1} \ldots e_k\}$; and

mons:$<e\ h> =$ if $i = 0$ then $<| e\ e_1 e_2 \ldots e_k\}$

else UNDEFINED.

Structured Application

Consider f:h when $f = <f_1\ f_2\ \ldots\ f_{i_0} | f_{i_0+1}\ \ldots\ f_{k_0}\}$

and let $h = <h_1\ h_2\ \ldots\ h_i | h_{i+1}\ \ldots\ h_n\}$ with

$h_j = <e_{j,1}\ e_{j,2}\ \ldots\ e_{j,i_j} | e_{j,i_j+1}\ \ldots\ e_{j,k_j}\}$

for $j>0$. Fix $k = \min\limits_{j \geq 0} k_j$.

Then $<f_1\ \ldots\ f_{i_0} | f_{i_0+1}\ \ldots\ f_{k_0}\}:h$ is defined to be

$$\begin{array}{l} <f_1:<1:h_1 \ldots 1:h_i | \ldots 1:h_n\} \\ \quad f_2:<2:h_1 \ldots 2:h_i | \ldots 2:h_n\} \\ \quad \vdots \\ \quad f_{i_0}:<i_0:h_1 \ldots i_0:h_i | \ldots i_0:h_n\} \;\Big| \\ \quad \vdots \\ \quad f_k:<k:h_1 \ldots k:h_i | \ldots k:h_n\}\ \} \end{array}$$

This protocol for structured functions is called *structured application* (elsewhere *functional combination* [12,16 ,17].)

As an example of structured application we present the function *deal* which partitions a finite sequence into two sequences, its even-numbered elements and its odd-numbered elements; much like a deal for a two handed game of Old Maid [17].

```
    deal:s ≡ if empty?:s then <□□>          ; two empty structures
             if empty?:rest:s then <s □>  ; a singleton sequence
             else <cons*>:<                ; deal two cards
                     s
                       deal:rest:rest:s > .
```

Since the base results of deal is always a pair, the recursion in the last line also produces a pair.

Suspending constructors play a critical role in twisting the order of evaluation of a program written as a recursive program to behave as iterative code [15]. The idea, related to Wadsworth's call-by-need [45] and to Vuillemin's call-by-delayed-value [44], is that every field in a Hoare [22] record structure is set only once as the node is created, but that initial contents need not be the ultimate value; a suspension [13] of that value will do. A suspension consists of a form and an environment, enough information for the evaluator to manifest that value on the first use of the contents of that field. Because the value is never computed unless it's needed, least fixed-point semantics result [40]. Because the value is never computed until it's needed the evaluation order and resource requirement is drastically altered [15]. Infinite sequences [14] (related to streams [27]) become available without the grief of coroutines [18,24], while applicative code written years ago suddenly lends itself to multiprocessing [16]. The concept of suspension will also be used in implementing multisets.

As an example of such infinite sequences expressed without coroutines consider <u>integers</u>, the value of ints:1 where

ints:i ≡ cons:<i ints:succ:i> .

Since the construction of the sequence is suspended, only as much of <u>integers</u> is calculated as is ever used directly or indirectly in determining the output of a program which includes this definition [15].

Not only are structures created only when necessary using suspensions, but also they are destroyed rather sooner than the programmer might have expected. The storage manager, in our implementation a reference count system, recovers the unneeded prefix of a structure whose suffix remains suspended pending eventual use. A snapshot at any time would show that the representation of what the user perceived as a structure (whose size is data dependent) might be, rather, represented by a suspended structure of constant size. Such is the case with <u>integers</u> above and the intermediate files in the n-way <u>merge</u> presented below. Input is a sequence of an arbitrary number of sorted files, which are merged pairwise into half the number of sorted intermediate files; pairwise merging recurses until there is only one final sorted file. Knuth [25] describes the space behavior using coroutines and programs the tree structure explicitly; we present code as if it depended on extensive intermediate results which never materialize.

```
merge:files ≡ if empty?:files then □        ;no more files
              if empty?:rest:files then first:files  ;only one file
              else mergetwo:<merge*>:
                                 deal:files  .
```

We now focus on the multiset, the new data structure described above. For our purposes it is perceived to be an unordered set which allows duplication (see also *bags* [46, 38]), but some items may be computationally undefined. Each item is initially a suspension which is coerced as part of a uniform (parallel) evaluation strategy when the *first* or *rest* of a multiset is *needed*. For the user the structure behaves much like a sequence which is unordered and, because of divergent elements, whose suffix may be inaccessible. For a single processor, such a strategy might be to take one step of the evaluation on each suspension of the multiset in turn until one converges to a value. On a multiprocessor system one could dispatch one processor on each suspension, let them run asynchronously, and wait for any one to converge; all other processors might then be released.

An interesting, yet simple, example which uses multisets is the symmetric <u>and</u> which terminates with FALSE, TRUE, or never terminates if all unresolved computations within the multiset diverge [33]. Consider

```
and:{zero?:arcsine:2  zero?:sub2:2  zero?:2}         (1)
and:{zero?:arcsine:2  zero?:arcsine:3  zero?:0}      (2)
and:{zero?:times:<2 0>  zero?:sub2:2  zero?:0}       (3).
```

Since zero?:2 is always FALSE, (1) evaluates to FALSE; (3) evaluates to TRUE because all items are TRUE; but (2) never terminates after it determines zero?:0 is TRUE.

Although the program for <u>and</u>, below, was originally written for the sequential <u>and</u> [31,13], if its argument is a multiset, it behaves as if it were the symmetric <u>and</u>.

```
and:h ≡ if empty?:h then TRUE
        if first:h then and:rest:h
        else FALSE  .
```

functions represented as a multiset, then its result is a multiset and that *integers* is discussed above. Second, the invocation of *atom*? in the program below requires that its argument actually be manifest [16]. Even if we understand that each element of a file is an atom and that such an invocation necessarily returns TRUE, An interesting, yet simple, example which uses multisets is the symmetric *and* which terminates with FALSE, TRUE, or never terminates if all unresolved computations within the multiset diverge [33]. Consider

```
and:{zero?:arcsine:2  zero?:sub2:2  zero?:2}        (1)
and:{zero?:arcsine:2  zero?:arcsine:3  zero?:0}     (2)
and:{zero?:times:<2 0>  zero?:sub2:2  zero?:0}      (3).
```

Since zero?:2 is always FALSE, (1) evaluates to FALSE; (3) evaluates to TRUE because all items are TRUE; but (2) never terminates after it determines zero?:0 is TRUE.

Although the program for *and*, below, was originally written for the sequential *and* [31,13], if its argument is a multiset, it behaves as if it were the symmetric *and*.

```
and:h ≡ if empty?:h then TRUE
        if first:h then and:rest:h
        else FALSE .
```

We next present Dijkstra's [11] guarded-if statement, but extended to allow that some of the predicates or values may be computationally divergent.

guardedif:$<p_1 e_1 \ldots p_n e_n>$ becomes

first:{guard:$<p_1\ e_1>$...guard:$<p_n\ e_n>$}

where

guard:<p e> ≡ <u>if</u> p <u>then</u> e

<u>else</u> UNDEFINED .

Evaluation of <u>first</u> over a multiset will continue until a satisfactory p_i and e_i both converge.

The final multiset example is an interrupt handler for, say, the teletype driver of an operating system. Others have assumed this as a primitive [35 ,26]. The invocation serialize:files takes as an argument a sequence of an arbitrary number of sequential files, each of which is being created at a different rate, and returns a sequence of ordered pairs. Each pair is composed of a character and an integer identifying the file from which it was extracted; the i^{th} file may be reconstructed by projecting out, in order, the first elements of all pairs resulting from serialize:files which have i as a second element.

Two observations are necessary to understand the following code. First, recall that when a structured application has its functions represented as a multiset, then its result is a multiset and that <u>integers</u> is discussed above. Second, the invocation of <u>atom</u>? in the program below requires that its argument actually be manifest [16]. Even if we understand that each element of a file is an atom and that such an invocation necessarily returns TRUE, nevertheless that predicate requires that its argument actually exist (it is strict [44,13]). We use it as a "pause until it's there".

```
serialize:files ≡ assemble:{process*}:<
                            files
                            integers > .

process:<file index> ≡ if empty?:file then <EOF UNDEFINED index>
                       if atom?:first:file
                         then <first:file rest:file index>
                       else UNDEFINED.

assemble:triples ≡ cons:<<1:first:triples 3:first:triples>
                         assemble:mons:<process:rest:first:triples
                                        rest:triples>>
```

The effect of *process* is to converge to a partitioning triple just as soon as the next item in a file becomes available; it may be a long wait.

Parallelism

In this section we summarize the provisions for parallel implementation of the programming model described above. Now that the cost of processors makes multi-processors financially feasible, machine architects need to know how to interconnect them (not a cheap trick in itself) so that the resulting machine will be usable--programmable to solve important problems. We relate our programming model to several classic parallel architectures, demonstrating the ease with which simultaneous evaluation may be impressed upon functions expressed without these machine designs in mind. Most notably, the model 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.

The feature of suspending cons 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 their 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 conflict and communication. 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's running.

Structured application offers two sorts of parallelism. The first is exemplified by the code for deal. In the definition for this function the recursion is linear down the last parameter, but at each recursion step each of the two developing results must be handled. Clearly the pieces of the ultimately final result can be handled by two concurrent processes. So a simple but bounded parallelism is provided depending on the size (k in the definition of structured application) of the result when all elements of the function structure are defined independently of the recursive definition in which it appears.

Another kind of parallelism results if the function being defined appears in the function structure. The coding of the function merge is an example of this. If k processors are allocated for computing the result of a structured application, and it 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 tree results because a single processor evaluating a function encounters an instance of structured application and becomes dormant while the k 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 k at any node, with dormant processes at all non-leaves, and with active processes only at the leaves. If a hybrid of functions has more than one recursive call in such a scheme then a very "bushy" process tree can result. For example, Hoare's Quicksort Algorithm [21] can be implemented so that every recursion can make use of a new processor. At the n^{th} level 2^n processors may be used. 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 interprocessor protocol. Upon interpretation of structured application the active process goes dormant and spawns k 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 [49].

The star notation used on an argument to a structured application merely denotes that the argument is to be shared by all k processors. When the function itself is a starred structure then application is implicitly homogeneous and a different sort of concurrency may be used for interpreting the function over an arbitrarily sized argument. This use of structured application is most similar to mapping functions [31, 30] and their generalization [12]. An example is the function dotproduct, sum:{product*}, in which all multiplications may take place concurrently. Due to the expression of the function structure with the star, the compiler can easily detect that the same operation will be performed on all objects in the linear structures which become arguments in the multiset. Then the evaluation may proceed using pipelining across the k arguments to the starred function structure.

The opportunity for parallelism in accessing a multiset (or a hybrid which has been reduced by repeated applications of rest to a multiset) should be apparent. On a single processor the evaluation effort of the function first must be distributed carefully over all suspensions of the multiset which is its argument. In a multiprocessing environment a processor may be responsible for evaluating suspensions of only a "sub-multiset" when the same effort is divided among several processors. Ideally these sub-multisets are each singletons so that the distributed evaluation becomes truly simultaneous asynchronous evaluation. The protocol for such a computation is completely removed from the programmer and is up to the implementation of the computation model in a particular (processor-rich or processor-poor) environment.

Conslusions

Now we turn away from the specific language structure which has been presented in the previous sections and return to the course of the project itself. To a certain extent the problems we shall consider are concerned with the use of these and other tools, but the issue of programming style plays a role as well. Tools must be well suited for use by programmers and programmers must be well disciplined in their use before the powers of both will work well together.

The fundamental question, again, which underlies this effort is whether it is practical to solve all problems through applicative programming. From the programmer's perspective the problem is one of expressiveness and style. Can he tolerate this restricted set of programming tools to form his ideas within their range of expression? What sort of guidance should he be given in his use of multisets, whose realization may require much of the computing resource? Should he be allowed to specify when simultaneous evaluation (e.g. "dragging" defined in [17]) should occur or can this decision be delayed until compile or run-time? Similar to this question are many others based on the knowledge of the execution environment required by the programmer; we intend that it be minimal. Ought the programmer to be able to anticipate effort or order of evaluation within a multiset? Should he be allowed to allocate resources among its components, whose evaluations converge to members of the multiset? Should he be allowed infinite multisets; is the resource allocation problem for infinite multisets anything less than that for self-reproducing automata [43]?

An issue intermediate between style and implementation is that of program verification. The already impressive results on proving applicative programs [3 ,4] will extend to our control structures. Although accessing a multiset is time dependent we shall be able to verify that the (nondeterministic) results are valid. A separate axiomitization will be necessary to describe the explicit time dependence of results (i.e. real-time behavior). Proving the time dependencies of output on input is a third issue for program verification newly raised by suspended evaluation; it is a problem which may be solved after the value of a computation is proved valid but before termination [33] is proved since partial or infinite computations may still yield output under such a protocol.

The value UNDEFINED, resulting from noticeably divergent computations (as opposed to resource exhaustion resulting from undetectably divergent ones) plays a special role in a multiset: it disappears. Should this value be propagated throughout the computational model as well? Is one undefined token sufficient, given the rigors of program development? What other tools are needed to debug new programs? How can one analyze an algorithm whose nondeterminate behavior is sensitive to run-time environment? What properties about the multiprocessing environment must be known to predict behavior?

Finally we turn to questions of implementation based on these tools. We have considered some possibilities for multiprocessing [16]; are there others? Should such a language be compiled (in the sense of PASCAL) [47] or should it be interpreted? Can feasible firmware or hardware be designed to facilitate a fast interpreter? [15] If so, can it be extended naturally to a system of many processors?

What new data structures are introduced in implementing specific algorithms? Storage is now being managed by a reference count scheme [7]; can it be used to effect in-place data manipulations where the programmer specifies "create and destroy"? Can the determination of "in place" behavior be moved back from run-time to compile-time? How does an external file system influence the course of evaluation within a program? We have specified macroscopic behavior [17] but specific interfacing depends very much on the operating system; can device/program connection be specified without rewriting the entire operating system? Another open data structure problem is definition of the "near metric" giving a "locus" of a processor within a data structure in a multiprocessing environment [16] without interfering (say, through communication) with the progress of that processor.

References

1. J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Comm. ACM* 21, 8 (August, 1978) 613-641.

2. J. deBakker. Least fixed-point revisited. *Theoretical Computer Science 2*, 2 (1976) 155-181.

3. R. Boyer and J.S. Moore. Proving theorems about LISP functions. *J. Assoc. Comput. Mach. 22*, 1 (January, 1975), 129-144.

4. W. H. Burge. *Recursive Programming Techniques*, Addison-Wesley, Redding, MA (1975).

5. R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *J. Assoc. Comput. Mach. 24*, 1 (January, 1977), 44-67.

6. A. Church. *The Calculi of Lambda Conversion* (*Ann. of Math. Studies 6*), Princeton Univ. Press, Princeton (1941).

7. G. E. Collins. A method for overlapping and erasure of lists. *Comm. ACM 3*, 12 (December, 1960), 655-657.

8. H. B. Curry and R. Feys. *Combinatory Logic I*, North-Holland, Amsterdam (1958).

9. J. B. Dennis. First version of a data flow language. In B. Robinet (ed.), *Programming Symposium*, Springer-Verlag, Berlin (1974), 362-376.

10. E. W. Dijkstra. Recursive programming. *Numer. Math. 2*, 5 (October, 1960), 312-318.

11. E. W. Dijkstra. *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ (1976).

12. D. P. Friedman and D. S. Wise. An environment for multiple-valued recursive procedures. In B. Robinet (ed.), *Programmation*, Dunod Informatique, Paris (1977), 182-200.

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

References (Cont.)

14. D. P. Friedman, D. S. Wise, and M. Wand. Recursive programming through table look-up. Proc. ACM Symp. on Symbolic and Algebraic Computation (1976), 85-89.

15. D. P. Friedman and D. S. Wise. Output driven interpretation of recursive programs, or writing creates and destroys data structures. Information Processing Lett. 5, 6 (December, 1976), 155-160.

16. D. P. Friedman and D. S. Wise. The impact of applicative programming on multiprocessing. IEEE Trans. on Comput. C-27, 4 (April, 1978), 289-296.

17. D. P. Friedman and D. S. Wise. Aspects of applicative programming for file systems. Proc. ACM Conf. on Language Design for Reliable Software, SIGPLAN Notices 12, 3 (March, 1977), 41-55.

18. D. P. Friedman and D. S. Wise. Unbounded computational structures. SOFTWARE-Practice and Experience 8, 2 (August, 1978), 407-416.

19. P. Henderson and J. Morris, Jr. A lazy evaluator. Proc. 3rd ACM Symp. on Principles of Programming Languages, 95-103.

20. C. E. Hewitt and B. Smith. Towards a programming apprentice. IEEE Trans. on Software Engineering SE-1, 1 (March, 1975), 26-45.

21. C. A. R. Hoare. Quicksort. Computer J. 5, 1 (April, 1962), 10-15.

22. C. A. R. Hoare. Recursive data structures. Internat. J. Comput. Information Sci. 4, 2 (June, 1975), 105-132.

23. G. Kahn. The semantics of a simple language for parallel programming. Proc. IFIP Congress 74, North-Holland, Amsterdam (1974), 471-475.

24. G. Kahn and D. MacQueen. Coroutines and networks of parallel processes. Proc. IFIP Congress 77. North-Holland, Amsterdam (1977), 993-998.

25. D. E. Knuth. Sorting and Searching, Addison-Wesley, Reading, MA (1973).

26. Paul Kosinski. Arbiters. Talk at Workshop on Data Flow and Reduction Languages, Univ. of California, Irvine (1977).

27. P. J. Landin. A correspondence between AIGOL 60 and Church's lambda notation. Comm. ACM 8, 2 (February, 1965), 89-101.

References (Cont.)

28. P. J. Landin. The next 700 programming languages. *Comm. ACM. 9*, 3 (March, 1966), 157-162.

29. B. M. Leavenworth. On the construction of nonprocedural programs. IBM Research Report RC 6155, Yorktown Hts. (1976).

30. L. A. Lombardi. On table operating algorithms. *Proc. IFIP Congress 62*, North-Holland, Amsterdam (1963), 511.

31. J. McCarthy, P. W. Abrahams, D. J. Edwards, T. P. Hart, and M. I. Levin. *LISP 1.5 Programmer's Manual*, M.I.T. Press, Cambridge, MA (1962).

32. J. McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg (eds.), *Computer Programming and Formal Systems*, North-Holland, Amsterdam (1963), 33-70.

33. Z. Manna. *Mathematical Theory of Computation*, McGraw-Hill, New York (1974), Chapter 3.

34. J. H. Morris, Jr. and B. Wegbreit. Subgoal induction. *Comm. ACM. 20*, 4 (April, 1977), 209-222.

35. S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Comm. ACM. 19*, 5 (May, 1976), 279-285.

36. W. V. Quine. *Word and object*, M.I.T. Press, Cambridge, MA (1960), 144.

37. J. C. Reynolds. Definitional interpreters for higher-order programming languages. *Proc. ACM National Conference* (1972), 717-740.

38. L. Robinson and K. N. Levitt. Proof techniques for hierarchically structured programs. *Comm. ACM. 20*, 4 (April, 1977), 271-282.

39. H. Rogers, Jr. *Theory of recursive functions and effective computability*, McGraw-Hill, New York (1967).

40. D. Scott. Logic and programming language. *Comm. ACM* 20, 9 (September, 1977), 634-641.

41. P. D. Summers. A methodology for LISP program construction from examples. *J. Assoc. Comput. Mach. 24*, 1 (January, 1977), 161-175.

42. G. J. Sussman and G. L. Steele. SCHEME: an interpreter for extended lambda calculus. Artificial Intelligence Memo 349, Mass. Inst. of Tech. (1975).

43. J. Von Neumann. *Theory of Self-Reproducing Automata*, Univ. of Illinois Press, Urbana (1966).

44. J. Vuillemin. Correct and optimal implementation of recursion in a simple programming language. *J. Comp. Sys. Sci. 9*, 3 (December, 1974), 332-354.

45. C. Wadsworth. *Semantics and Pragmatics of Lambda-calculus*, Ph.D. dissertation, Oxford (1971).

46. R. J. Waldinger and K. N. Levitt. Reasoning about programs. Artificial Intelligence *5*, *3* (Fall, 1974), 235-316.

47. Mitchell Wand and Daniel P. Friedman. Compiling lambda expressions using continuations and factorizations (revised July, 1977). To appear *Journal of Comp. Lang.*

48. B. Wegbreit. Mechanical program analysis. *Comm. ACM 18*, 9 (September, 1975), 528-539.

49. 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. *SIGPLAN Notices 12*, 5 (May, 1977), 14-15.

Resources

The principal computing resource for this grant will be the Control Data 6600/CYBER 72 at the Indiana University Wrubel Computing Center because of the excellent PASCAL compiler available. Language implementations will be ultimately programmed in PASCAL. Most program design and maintenance will be done on the university's Digital Equipment DEC-10 in Indianapolis, which has superior editing systems for code maintenance and interactive LISP for designing the computational model.

The I.U. Computer Science Department has an active hardware design group with a laboratory equipped to fabricate plated-through boards. The faculty members and technicians from this group will provide insight and engineering advice on matters of parallelism relating to hardware capabilities and constraints.