An execution architecture for FP

by Tien Huynh
Brent Hailpern
Lee W. Hoevel

FP is a functional programming language proposed by John Backus to liberate programming from the "von Neumann style." We define here an architecture (FP machine model) that is intended to allow easy and efficient implementation of FP on a conventional von Neumann machine. We present a new execution architecture for FP based on Johnston's contour model and on Hoevel and Flynn's notion of DEL/DCA architectures. We call our architecture DELfp, a Directly Executed Language for FP.

1. Introduction

FP, the functional programming language proposed by John Backus, comprises a set of objects, a set of primitive functions, a set of functional forms, and an "application" operator. In FP, all programs are functions built from existing functions using the functional forms. There are no variables in an FP program. All functions map objects into objects and always take a single argument. Details of the FP syntax can be found in Backus's Turing Award lecture [1].

Following Backus's proposal, a number of architectures have been designed to execute functional languages [2]. Our goal was to define an architecture that would permit us to implement FP easily and efficiently on a conventional von Neumann machine. Our execution architecture is based on Johnston's contour model [3] and on Hoevel and Flynn's notion of DEL/DCA architectures [4, 5].

Our architecture is called DELfp for Directly Executed Language for FP. DELfp is both a language-sensitive and a host-sensitive execution architecture. DELfp is language-sensitive in that it embodies *all* of the semantic content of the source language. Indeed, the structure of DELfp parallels the structure of FP. The operator codes (opcodes) in DELfp instructions are functional forms[1] rather than functions. The arguments are functions rather than data objects. FP functions map sequences of objects (atoms and sequences) into sequences of objects. FP has no variables or global data storage. Similarly, there are no locations, data address offsets, or variables mentioned explicitly in a DELfp program.

However, DELfp is also designed with the host machine in mind. In our experiment the IBM PC AT is the host machine. Information as to the intent, restrictions, and structure of the original source program is encoded in a manner that minimizes the data transfers needed to execute a DELfp program. This objective follows from our presumption of a memory-limited host with multiple levels of storage hierarchy, each of which must be used in an optimal manner.

We divide DELfp programs into two categories: *external* and *local*. User-defined functions are translated into *external* programs. *Local* programs correspond to the predicate or consequence parts of a conditional functional form, the applied part of an apply-to-all functional form, or the inserted part of an insert functional form in the FP source program. The only difference between an external program and a local program is that external programs have symbolic names, and therefore can be referenced by other external programs. Local programs can only be referenced within the program in which they are defined.

[1] A functional form (or functional or combining form) takes functions or values as arguments and produces a function as a result. For example, the composition $f \cdot g$ takes f and g as arguments and results in a new function $f(g(x))$.

Table 1 Comparison of DELfp with conventional architecture.

	Operator	*Operand*	*Execution*	*Control*
Conv. ISA	ALU functions: i.e., +, −, ×, ÷	Storage cells	Sequence of states	Explicit operators
DELfp	Functionals: i.e., if, apply, insert	Functions: primitive or user-defined	Sequence of functional computation	Implicit in formats

DELfp is in close correspondence with the source high-level language. Hence, DELfp programs can be converted back to the FP source syntax simply by one pass of translation. This is in contrast to the traditional *lack* of correspondence between von Neumann instruction set architectures and conventional high-level languages. We find untranslation to be an elegant technique and one that avoids the fix-but-don't-recompile syndrome. We have only one version of a function at a time.

DELfp and a conventional architecture are similar in the following ways:

- Both consist of a sequence of syllables (bytes or words).
- Both have formats binding operands to an operator.
- Both use the general sequencing rule of increasing the instruction pointer.

DELfp differs from a conventional architecture in the following ways:

- DELfp operators are functional forms rather than ALU functions.
- DELfp operands are primitive or user-defined functions rather than storage cells.
- DELfp execution computes the value of a functional rather than changing a global state.
- DELfp control is encoded implicitly in the formats rather than employing explicit operators. There are *no* explicit jump operators in DELfp.

These comparisons are broadly summarized in **Table 1**. DELfp is defined by

- An instruction set,
- A set of residual control variables, describing the interpretive environment at any execution point (for example, one residual control variable is the instruction pointer),
- A contour model [3] for retention of activation records, and
- A set of format rules to determine the location of data manipulation during each computation step.

We now examine the details of this definition, as well as a prototype DELfp implementation consisting of a compiler, interpreter, and decompiler.

2. DELfp instructions

Each DELfp instruction corresponds to an FP functional. DELfp instructions are encoded byte strings in the DELfp program space. All instructions are partitioned into distinct syllables [6], each of which is one byte long. The leading byte is always the format syllable. Therefore, the length of each instruction is at least one byte, but may vary up to four bytes.

The remaining bytes, if any, are operand syllables, each of which may be one of five different types:

- A pointer to a primitive function.
- An indirect pointer to the address of an external program.
- An integer in the range from 0 to 255.
- A pointer to the constant object table.[2]
- An indirect pointer to the address of a local program.

The interpretive mechanism recognizes the type of the operand syllable (if any) once the format syllable has been decoded.

3. Residual control variables

DELfp semantics can be modeled using four residual control variables to contain the current state of a computation.

The environment pointer (*ep*) identifies the active contour; it points to the logical base of the architecture contour, and all entities in the active contour are identified as offsets from *ep*. In our implementation, contours are kept in a LIFO stack; this is a direct result of the function call/return semantics of FP. Data are not stored in the contour, but rather in an object space to which the contour can point.

The instruction pointer (*ip*) is used to locate an execution point in the instruction stream. When a contour is entered, the current value of *ip* is saved, and *ip* is set to point to the first instruction of the program. On leaving a contour, *ip* is set to the saved value of the instruction pointer.

The source pointer (*sp*) specifies the input location for the next function evaluation. It may be a pointer to a parameter passed from the calling function, or to the address within this contour of a temporary variable pointing to an intermediate result. The destination pointer (*dp*) specifies the output location for the result of an evaluation. Together, *sp* and *dp* are the residual control for identifying the DELfp data stream.

Together, these four control variables identify a particular point of execution and all the functions and objects accessible at that point of execution.

[2] We do not express numeric and character constants in line. That would result in large instruction formats. Instead we place constants in a table. Instructions, then, index into the table for constant operands. This generalizes easily to constant sequences and trees, which would be impractical in the instruction stream.

4. DELfp contour

We use a contour model,[3] similar to that developed by Johnston [3], to describe the storage mapping transforms required by the allocation, release, and retention rules of DELfp. We chose this model because of its generality and simplicity. DELfp passes parameters "by reference" simply by copying the appropriate descriptors from the caller's argument slots into the callee's argument slots. We differ from Johnston in that we do not trace through layers of contours to resolve a reference [7]. Each cell in our contour points to the data or to the location for a return parameter.

The layout of an active contour is shown in **Figure 1**.

In DELfp, a skeleton contour is associated with each program definition. The skeleton contour describes the starting address of the program and the number of temporary variables needed to execute the program. When a program is called, a new contour is created. The first entity of the contour indicates the minimum number of temporary variables needed to execute the program. Hence, the value of the first entity specifies the number of stack cells that are reserved for the contour. The first six entities of the contour are fixed for all programs, but the number of temporary variables may vary. The final destination pointer holds the address within the calling function where the result of the program will be stored.

If the current program calls another program, the interpretive mechanism creates a new contour. Therefore, it is necessary to save the next instruction's address and the source and destination pointers (intermediate values) in the continuation cells. Note that the cell that is used for storing the saved value of the source pointer initially holds the address of the input argument, so it is also an argument slot. Also note that tail-recursive calls can be detected at compile time, permitting us to chain together the returns from nested user-defined functions.

Note that the value of *ep* from the calling function is not saved in the contour. Our LIFO implementation permits us to compute the value upon contour exit.

The residual control flag is used only when a conditional functional form is encountered. It is used to determine whether the predicate or consequence part should be executed.

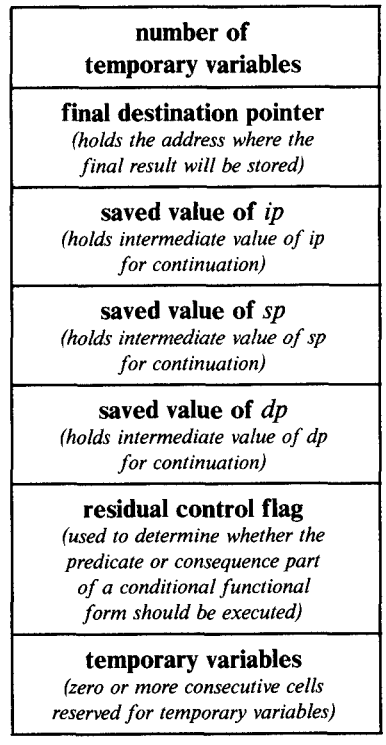

Figure 1

Layout of an active contour.

5. Formats

In DELfp, the format syllable specifies the meanings of the subsequent syllables, determines the size of the instruction, and indicates which operator is to be applied. It also provides the context so that the interpreter can deduce the appropriate temporary locations of the input and output data streams. Since DELfp is built on a von Neumann-style workstation (IBM PC AT), we cannot avoid allocating temporary memory locations to store the intermediate results of each computation. However, the allocation of these temporary variables can be done by decoding the instruction formats, hence freeing the DELfp instructions from explicitly introducing variables.

The format syllable is divided into two half-byte fields. The leading field is called the context mode, and it specifies the context of an instruction. We use seven designations to describe the context modes.

I: instruction is inside a construction unit,
B: instruction is at the beginning of a composition unit,
M: instruction is at the middle of a composition unit,
E: instruction is at the end of a composition unit,
R: instruction is at the end of the program,
L: instruction is at the end of the program and the end of a composition unit,
N: instruction has no effect on the context,

where a construction unit is of the form $[f, g, h, \cdots]$ and a composition unit is of the form $[f \cdot g \cdot h \cdot \cdots]$.

[3] A contour is an architectural abstraction that shares many of the aspects of an *activation record*, *frame*, or *stack window*. We do not use the full power of contours in DELfp, but this work is derived from other architectural work that does. Hence, we have made a conscious decision to cite the historical roots of our memory model in [3].

Table 2 Rules for determining temporary locations for the input and output data streams.

	APF, AUF, ALS, ARS, ACF, COND, ALPHA, INSERT		ENDCONS
	Before execution	After execution	Make sequence from *D..Dstk[top]*
I	*D = D + 1*		*D = Dstk[top]* *pop Dstk*
B	*push S on Sstk* *D = D + 1*	*S = D*	*D = Dstk[top]* *pop Dstk* *push S on Sstk* *S = D*
M			*D = Dstk[top] −1* *pop Dstk*
E		*S = Sstk[top]* *pop Sstk*	*D = Dstk[top] −1* *pop Dstk*
R	*D = O*		*D = O* *pop Dstk*
L	*D = O* *pop Sstk*		*D = O* *pop Dstk* *pop Sstk*

The second field of the format syllable is the operator. It specifies the functional form and the meaning of the subsequent syllables. There are ten such operators in DELfp. Their mnemonic codes are described as follows:

APF Apply Primitive Function: This applies a primitive function to the input data object. Its operand is a pointer to the primitive function being applied.

AUF Apply User-defined Function: This applies a user-defined function to the input data object. Its operand is a pointer to the user-defined function symbol table, which in turn points to the starting address of that function in the program space.

ALS Apply Left Selection: This selects the *n*th element from the left of the input data object, where *n* is encoded in the subsequent syllable as the operand.

ARS Apply Right Selection: This operator has the same function as ALS except that the selection is from the right.

ACF Apply Constant Function: This returns a constant object. Its operand is a pointer to the constant object table containing the value to be returned.

COND Condition: COND takes three operands, which point to the starting address of the predicate (IF part), the positive consequence (THEN part), and the negative consequence (ELSE part). Each part is represented by a local program.

ALPHA Apply to all: This applies the operand function to each element of the input stream. The operand function is a local program.

INSERT Insert: This inserts the operand function between each element of the input stream. The operand function is a local program.

BEGCONS Begin Construction: This indicates that a construction is encountered. Its operand is an integer indicating the depth of consecutive nested "]" in the FP source text.

ENDCONS End Construction: This operator concludes a construction. It links all the results produced by each component inside that construction into one sequence. This operator does not have an operand.

Note that the BEGCONS and ENDCONS operators are not symmetric because the context mode of the BEGCONS operator is irrelevant to the interpretive mechanism; therefore, we can encode nested BEGCONS ("]") into one instruction. We treat ALS and ARS as functionals instead of as primitive functions because they both need one more byte than primitive functions to encode the position of the desired element in the input sequence object.

As mentioned before, each DELfp instruction has a format syllable, and hence each instruction has a context mode. The rules for assigning the context mode to an instruction are

1. An instruction with the BEGCONS operator always has an N context mode.
2. The last instruction of an external or a local program has an R context mode if it is not bound to a composition unit.
3. The last instruction of an external or a local program has an L context mode if it is bound to a composition unit.
4. Any instruction that is bound to a composition unit has one of the B, M, or E context modes, depending on whether it is the beginning, middle, or ending component of that composition unit.
5. Any instruction that is inside a construction but not bound to any composition unit has an "I" context mode.

In **Table 2** the rules are presented for deducing the appropriate temporary locations for the input and output data streams from the formats. *S* (source), *D* (destination), and *O* (final destination) are all indexes of object storage locations in the contour. The variables *Sstk* and *Dstk* represent two global stacks used to save the *S* and *D* values during intermediate computations.

The semantics of the ENDCONS instruction is to form a sequence from the objects pointed to by stack locations $D, D-1, \cdots, Dstk[top]$ (the value of *D* is obtained from the prior instruction). The resulting sequence is then stored in

the location pointed to by the D derived from the table. When we say "form a sequence," we assume that the architecture can create a new compound object (sequence) out of a collection of objects (atoms or sequences).

The semantics of the BEGCONS instruction is to push $D + K$ on *Dstk*, where K is the argument to the instruction. BEGCONS always takes context mode N.

6. An example

Having the context modes, operators, and rules for assigning context modes, we can now show how DELfp instructions are generated from FP programs. Suppose we are given user-defined functions f_1, f_2, f_3, f_4 and primitive functions p_1, p_2, p_3. Let F1, F2, F3, F4 be pointers to the symbol table for f_1, f_2, f_3, f_4 and P1, P2, P3 be pointers to primitive functions p_1, p_2, p_3. The FP program $f_1 \cdot [p_1, f_2, p_2 \cdot [f_3, f_4]] \cdot p_3$ is compiled into sequences of DELfp instructions:

B	APF	PE
N	BEGCONS	2
I	AUF	F4
I	AUF	F3
B	ENDCONS	
E	APF	P2
I	AUF	F2
I	APF	P1
M	ENDCONS	
L	AUF	F1

To illustrate how storage cells can be deduced from the formats, we use the same example. Let $S = 0$ and $O = 1$; that is, the input data object for the program is stored in location $L[0]$ and the output, when finished, should be stored in $L[1]$. Our goal is to derive the proper indices for the source and destination in each instruction to match the following sequences of functional computation:

1. Apply p_3 to object in $L[0]$ and store the result in $L[2]$.
2. Apply f_4 to object in $L[2]$ and store the result in $L[3]$.
3. Apply f_3 to object in $L[2]$ and store the result in $L[4]$.
4. Make a sequence from objects in $L[4]$, $L[3]$ and store it in $L[3]$.
5. Apply p_2 to object in $L[3]$ and store the result in $L[3]$.
6. Apply f_2 to object in $L[2]$ and store the result in $L[4]$.
7. Apply p_1 to object in $L[2]$ and store the result in $L[5]$.
8. Make a sequence from objects in $L[5]$, $L[4]$, $L[3]$ and store it in $L[2]$.
9. Apply f_1 to object in $L[2]$ and store the result in $L[1]$.

We now go through each instruction of the program and record the changes to S and D. Initially, D is set equal to O, and both *Sstk* and *Dstk* are empty.

Instruction: B APF P3
Actions:
push S on *Sstk* → $Sstk[top] = 0$
$D = D + 1$ → $D = 2$
apply p_3 to object in $L[S] \equiv L[0]$
and store result into $L[D] \equiv L[2]$
$S = D$ → $S = 2$

Instruction: N BEGCONS 2
Actions:
push $D + 1$ on *Dstk* twice → $Dstk[top] = 3$

Instruction: I AUF F4
Actions:
$D = D + 1$ → $D = 3$
apply f_4 to object in $L[S] \equiv L[2]$
and store result into $L[D] \equiv L[3]$

Instruction: I AUF F3
Actions:
$D = D + 1$ → $D = 4$
apply f_3 to object in $L[S] \equiv L[2]$
and store result into $L[D] \equiv L[4]$

Instruction: B ENDCONS
Actions:
form a sequence from objects
in $L[D..Dstk[top]] \equiv L4..3]$
$D = Dstk[top]$ → $D = 3$
pop *Dstk* → $Dstk[top] = 3$
store the resulting sequence into $L[D] \equiv L[3]$
push S on *Sstk* → $Sstk[top] = 2$
$S = D$ → $S = 3$

Instruction: E APF P2
Actions:
apply p_2 to object in $L[S] \equiv L[3]$
and store result into $L[D] \equiv L[3]$
$S = Sstk[top]$ → $S = 2$
pop *Sstk* → $Sstk[top] = 0$

Instruction: I AUF F2
Actions:
$D = D + 1$ → $D = 4$
apply f_2 to object in $L[S] \equiv L[2]$
and store result into $L[D] \equiv L[4]$

Instruction: I APF P1
Actions:
$D = D + 1$ → $D = 5$
apply p_1 to object in $L[S] \equiv L[2]$
and store result into $L[D] \equiv L[5]$

Instruction: M ENDCONS
Actions:
from a sequence from objects
in $L[D..Dstk[top]] \equiv L[5..3]$
$D = Dstk[top] - 1$ → $D = 2$
pop *Dstk* → *Dstk* is empty
store the resulting sequence into $L[D] \equiv L[2]$

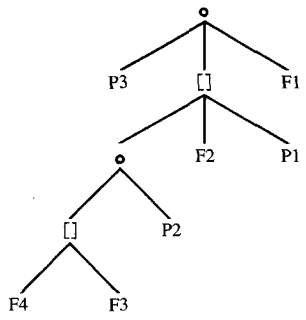

Figure 2

Abstract syntax tree.

Instruction: L AUF F1
Actions: $D = O$ $\rightarrow D = 1$
apply f_1 to object in $L[S] \equiv L[2]$
and store result into $L[D] \equiv L[1]$
pop *Sstk* $\rightarrow$ *Sstk* is empty

7. Compilation

Our compiler translates the FP source text into DELfp programs using a two-phase process. In the first phase, we employ an LR parser to parse the FP source text and generate its abstract syntax tree. For instance, the last example compiles into the abstract syntax tree shown in **Figure 2**.

Here "·" and "[ ]" are canonical forms of a composition unit and a construction unit, respectively. Both composition and construction are *n*-ary operators in the parse tree. We implement this by conventional parent/first-sibling and sibling/next-sibling links.

In the second phase, the abstract syntax tree is traversed to generate the DELfp instructions using the following algorithm:

```
Generator:
  begin
    GenDEL(abstract_syntax_tree)/* generate main body
                                    of external
                                    program*/
    While queue is not empty do
      GenDEL(tree_on_top_of_queue)/* generate local
                                     program*/
    end
  end
```

where GenDEL is defined as

```
GenDEL(root):
  begin
    If root < > nil then
      begin
        If root is a CONDITION, ALPHA, or INSERT
        Then put root's subtree(s) on queue to be a local
        program
        Else GenDEL(root's_leftmost_subtree)
        Convert root to corresponding DELfp instruction
        GenDEL(root's_right_subtree(s))
      end
  end
```

8. Interpreter

Our interpreter is implemented on a partially mapped host (universal host). During interpretation, each syllable of the execution instruction stream is first decoded and then executed sequentially. The interpretive mechanism can be viewed as cycles of the following operations:

1. Decode Format—extract the leading syllable, and deduce the source and destination addresses.
2. Decode Operands—extract the operand syllable(s), if any. Bind operand(s) with the operator and transfer control to the appropriate semantic routine.
3. Execute—perform the designated semantic action, and store the result.
4. Housekeep—rearrange source or destination address if needed, and begin another cycle of interpretation.

In DELfp, the instructions can be easily fetched and decoded. There are no complicated residual controls during execution time, and there are no branch instructions. As a result, the interpretive mechanism is simple and suitable for implementation in software, microcode, or hardware.

9. Decompilation

As we mentioned earlier, because of the close correspondence between DELfp and FP, DELfp codes can be converted back to FP codes without much extra effort. The following recursive algorithm accepts the starting address of a DELfp program and converts it back to an FP program:

```
GenFP(starting_address):
  begin
    Let ip = address of first instruction with an R or L
    context mode
    Repeat
      Convert instruction pointed to by ip to corresponding
      FP notation
      If instruction calls other local program(s)
      Then GenFP(starting_address_of_local_program)
      Reduce ip to point to address of instruction prior to
      current one
    Until ip < starting_address
  end.
```

Using this algorithm, the source text that the user types need not be retained after parsing. In order to retrieve the exact image of the source codes, however, the interpreter must keep an "alter table" specifying details of indentation, spacing, and capitalization.

10. Comparison

A worthwhile comparison to other implementation strategies [2] is beyond the scope of this paper. We plan to make such a comparison in the future. We also believe it is premature to make execution comparisons with other systems, because of the lack of large FP benchmarks and the different implementation vehicles (for examples, PCs, VAXs, and meta-interpreters in LISP on various workstations).

We can, however, describe our implementation as a first step towards such a comparison. Our prototype editor/interpreter source consists of 4000 lines of Turbo Pascal code. This translates to a ".com" file of 43K bytes. We give two examples from [1], factorial and matrix multiply, which have the following FP syntax:

$$\text{Def } ! \equiv \text{eq} \cdot [\text{id}, \overline{0}] \rightarrow \overline{1}, \times \cdot [\text{id}, ! \cdot - \cdot [\text{id}, \overline{1}]]$$

$$\text{Def MM} \equiv (\alpha\ \alpha\ \text{IP}) \cdot (\alpha\ \text{distl}) \cdot \text{distr} \cdot [1, \text{trans} \cdot 2]$$

where

$$\text{Def IP} \equiv (/+) \cdot (\alpha \times) \cdot \text{trans}$$

The size of the corresponding DELfp programs in bytes and the running speed (Given in clock-on-the-wall seconds) on a PC AT (6 MHz) are shown in **Table 3**.

Table 3 Size versus running time on a PC-AT (8 MHz).

Example	*Size* (bytes)	*Speed* (s)
Factorial of 10	30	0.05
Matrix multiply (MM + IP) (10 by 10) (10 by 10)	50	3.3

11. Summary

We have presented a new execution architecture for FP. In our model, we emphasize the one-to-one correspondence between source- and execution-level entities. Keeping the execution architecture as close to the source as possible makes the translation and untranslation simple and natural. But even though translation is simple, the intermediate architecture is designed so that interpretation is still efficient. Furthermore, an execution architecture with high transparency ensures that program portability and compatibility need be assumed only at the source level. In our future work, we will investigate other options in format or opcode encoding and extend the DELfp to FFP [1] or FP84 [8].

Acknowledgment

We would like to thank the three anonymous referees for their help in revising an earlier draft of this paper.

References

1. John Backus, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," *Commun. ACM* **21,** No. 8, 613–641 (August 1978).
2. Steven R. Vegdahl, "A Survey of Proposed Architectures for the Execution of Functional Languages," *IEEE Trans. Computers* **C-33**, No. 12, 1050–1071 (December 1984).
3. J. B. Johnston, "The Contour Model of Block Structured Processes," *SIGPLAN Notices* **6**, 55–82 (February 1971).
4. Michael J. Flynn and Lee W. Hoevel, "Measures of Ideal Execution Architectures," *IBM J. Res. Develop.* **28,** No. 4, 356–369 (July 1984).
5. Michael J. Flynn and Lee W. Hoevel, "Execution Architecture: The DELtran Experiment," *IEEE Trans. Computers* **C-32**, No. 2, 156–175 (February 1983).
6. Michael J. Flynn, John D. Johnson, and Scott P. Wakefield, "On Instruction Sets and Their Formats," *IEEE Trans. Computers* **C-34**, No. 3, 242–254 (March 1985).
7. Lee W. Hoevel, "Directly Executed Language," Ph.D. Thesis, The Johns Hopkins University, Baltimore, MD, 1978.
8. Joseph Y. Halpern, John H. Williams, Edward L. Wimmers, and Timothy C. Winkler, "Denotational Semantics and Rewrite Rules for FP: Preliminary Version," *Research Report RJ-4245*, IBM San Jose Research Laboratory, San Jose, CA, December 1984.

Received June 27, 1985; accepted for publication June 5, 1986

Tien Huynh *IBM Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598.* Mr. Huynh received his B.S. and M.S. in computer science from Kent State University, Ohio, in 1983 and 1985. He is a senior associate programmer in the experimental languages project at the IBM Thomas J. Watson Research Center, where he has worked since 1984. His interests include functional languages and computer architecture.

Brent Hailpern *IBM Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598.* Dr. Hailpern received his B.S. in mathematics from the University of Denver, Colorado, in 1976, and his M.S. and Ph.D. in computer science from Stanford University, California, in 1978 and 1980. Dr. Hailpern is the senior manager of the experimental languages and concurrent systems projects at the IBM Thomas J. Watson Research Center, where he has worked since 1980. His research interests include concurrent programming, programming languages, and program verification. Dr. Hailpern is a member of the Association for Computing Machinery and a senior member of the Institute of Electrical and Electronics Engineers.

Lee W. Hoevel *National Cash Register, Dayton, Ohio.* Dr. Hoevel received his B.A. in mathematics and economics from Rice University, Houston, Texas, in 1968 and his Ph.D. in electrical engineering from The Johns Hopkins University, Baltimore,

Maryland, in 1979. In 1968, he joined the staff of The Johns Hopkins Applied Physics Laboratory, Silver Spring, Maryland. He took a leave of absence in the fall of 1972 to become a full-time graduate student of The Johns Hopkins University. He was a Research Assistant with the Department of Electrical Engineering, Stanford University, Stanford, California, from 1975–1979. Dr. Hoevel was a member of the technical staff at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, from 1978–1985. He is currently Director of the Advanced System Architecture Program at NCR World Headquarters, Dayton, Ohio. His current research interests include distributed programming, user interfaces, and system architecture.