F-code and its Implementation: a Portable Software Platform for Data Parallelism

V. B. MUCHNICK, A. V. SHAFARENKO AND C. D. SUTTON
Department of Electronic and Electrical Engineering, University of Surrey, GU2 5XH, UK

Received July 1993; revised September 1993

1. WHY PORTABLE SOFTWARE PLATFORMS (PSPs)?

The important questions for parallelizing compilers are: (i) Can enough parallelism be extracted from a program to permit a compiler to produce executable code *congruent* (Skillicorn, 1991) with a parallel machine architecture? (ii) Can a program be portably efficient: can a program be compiled congruently to any general purpose parallel machine? (iii) What is a general purpose parallel machine, or general purpose intermediate interface to machines; what is a general purpose abstract machine? This paper deals with these three issues by discussing PSPs. A PSP is an intermediate level for compilers; the particular PSP discussed in this paper, F-code, is a PSP for computational imperative programming languages. The intention of PSP is not unlike the intention of HPF (HPF, 1992), Fortran D (Fox *et al.*, 1992), Vienna Fortran (Zima *et al.*, 1992), etc., which is to foster an architecture-independent programming model in the parallel domain.

The nature of the design of programming languages is two-fold: any language is a programming paradigm which is intentionally well suited for expression of algorithms in some applications area; also a programming language is a semantical system in its own right, which implies that a program has a certain *meaning* irrespective of what the user wanted it for. The latter allows implementation of a language on a computer with some degree of efficiency, which is greater the more adequate the semantical system of the language is for the computer architecture.

Normally these two aspects of the programming language are not decoupled in the language design. For example, by providing array constructs Fortran 90 (Fortran 90, 1991) provides *both* the methodology of array processing (through the use of array extensions to Fortran operators and the set of built-in functions) and its conceptual basis (contained in the semantical definitions of those extensions and functions). The former is unique and reflects the Fortran syntactical tradition acknowledged by Fortran users, whereas the latter is found in any other data-parallel language, notably APL (Iverson, 1962) which was the first to introduce array processing facilities, APL2 (Brown *et al.*, 1988) and MOA (Hains and Mullin, 1993) based on the Bird–Meerten formalism (Bird, 1987, 1989).

Now that parallel processing has come of age, one ought to distinguish between different semantical systems in order to determine an appropriate hardware platform for their implementation. It is important to separate the conceptual basis of a language, about which one can reason in terms of hardware adequacy, from the language appearance, which pertains to the strategies of program design, and after all, traditions.

This leads us to understanding of the role of portable, or architecture-neutral, software platforms (PSP). A PSP is a practical model for parallel computation and must possess:

- High-level semantics including all sorts of abstractions inherent in high-level languages such as lists, tuples, pointers, data-types, etc. In a recent study (Marino and Succi, 1989) the requirements for useful parallel data structures were enumerated: data structures should: (i) match existing and future computer architectures, (ii) allow for efficient parallel implementations, (iii) be formally defined in an applicative language and (iv) allow the definition of complex objects in a constructive way. One feature of today's languages and parallelizing compilers is their emphasis on arrays. Scientific programmers are used to working with that data structure, but in many cases it is not the most appropriate for their needs. Some properties of object-oriented languages are necessary: perhaps just simple classes. A PSP must possess a type system that is interprocedurally verifiable as type-safe at compile-time.
- Primitive syntax making it easier for the top-level compiler to generate code. What needs to be expressed is data-dependencies at the data-parallel level where dependencies explicitly represent possible functional concurrency. It must possess *descriptive simplicity* (Skillicorn, 1991) reducing the overhead of describing and managing massive parallelism. Three kinds of abstractions exist for descriptive simplicity: (i) abstractions for decomposition or the explicit expression of parallelism, (ii) abstraction from the details of communication and (iii) abstraction from the details of synchronization. Skillicorn (1991) reviewed models for parallel computers and suggested that the best way to cope with these three kinds of abstractions is to use data-parallelism: a single-threaded SIMD style approach. However data-parallelism may restrict the forms in which com-

putation may be expressed, unless it is loosened to a multi-threaded concurrency with a predominantly breadth-first evaluation.

- *Architecture-neutrality.* There should be no assumption about specific hardware features, however the compiler must produce code of roughly equal efficiency to a machine-specific compiler. A PSP is a purely virtual machine which does not enforce any programming methodology. It is not a low-level specification of the algorithm but a representation of the application. Since it is architecture-neutral, a PSP compiler must be able to infer suitable distributions and re-distributions using interprocedural analysis for efficient parallelized execution on distributed memory machines, or the PSP must include necessary distribution directives. At this moment in time, automatic parallelization is thought to be impossible. Thus the MIMDizer (PSRC, 1991) and SUPERB (Zima *et al.*, 1988) were interactive semi-automatic systems.
- It should reflect at the model level an estimate of the cost of the underlying execution (congruence): performance should be appraisable to a reasonable accuracy in order to choose one compiler optimization over another during targetting the intermediate language to the machine in order to produce the most efficient solution.

A known PSP is a TDF (DRA, 1991) however, in its original form this platform was intended only for the targetting of languages to scalar machines. As such it is a scalar PSP. Recent research attempts to vectorize TDF (Lake and Sloman, 1992), however this inherits all of the bad features of vectorizing Fortran 77 programs. Therefore a PSP should automatically be an explicit data-parallel environment.

A PSP is an intermediate language for parallelizing compilers for all manner of high-level languages. The best way to represent data-parallelism is using a data-parallel algebra along the lines of a Bird–Meerten-like formalism. These formalisms give the programming language developer the ability to create a sufficiently rich assortment of data-parallel operators. Data-parallel algebra can be congruent if the operators are chosen sensibly such that they can reasonably be implemented (which may imply that the operators have to be low level).

The PSP in this document is different in that it is a high-level specification of the algorithm which can be evaluated lazily in array space, using selective assignment. It is a kind of formalism-independent high-level formalism whose intention is optimisation in an environment with obvious descriptive simplicity, and formally defined semantics [not least because it would need to be formally defined for automatic prototyping compiler–generators (Chen and Cowie, 1992)].

Its control semantics are not unlike Lisp (McCarthy, 1960). In particular, dynamic visibility of atoms and the explicitly introduced notion of association are exactly as in Lisp. The major difference is that control is transferred using a token passed to a SEQ/PAR control combinator rather than using labels and GOTO 'functions'.

A PSP decouples all methodologies from the program semantics and machine architecture thus enabling the top language compiler to concentrate mainly on the analysis of the source program. Due to the high level of abstraction the PSP adopts, the code generation from the top level language is easy and, perhaps more importantly, there is no need for optimization by the language compiler as it can be performed by the PSP compiler at the stage of targetting the code to a specific hardware platform. Besides, most of it can be done in a reflective fashion, i.e. from PSP back to PSP and therefore does not have to be re-done when changing the hardware, such optimization includes evaluation of common subexpressions, loop invariant elimination, vectorization of scalar code where possible, etc. It should be emphasized that the PSP representation of the source program is so close semantically to the source language that it is nearly reversible, i.e. it is possible to restore, with a good accuracy, the latter given the former. On the other hand the PSP can match a number of top languages, since for all their syntactical distinctions, these are not so different semantically.

2. BASIC CONCEPTS OF F-CODE

2.1. Concurrency

F-code is a PSP for parallel processing. Primarily it expresses data parallelism through operations on non-scalar objects. For example, this means that conceptually adding a matrix to a matrix is not a sequential loop in F-code, but rather an action performed on all matrix elements simultaneously.

The novelty of the F-code approach to data parallelism is that it supports the whole conceivable variety of parallel operations by providing a rank coercion mechanism, which allows one to apply parallel operators to any combination of objects, no matter if they have matching ranks or not. This is done in a regular way and involves just one notion of operand *orientation.* Also, a reasonable default is provided for the case of shape mismatch. Wherever the ranks are same or coerced F-code supplies a shape coercion rule.

Another sort of concurrency that F-code aims to accurately represent is the concurrency of evaluating the operands to an operation. It may be called *functional* parallelism, since this property is inherent in any functional programming language. For example, the Lisp expression.

```
(TIMES (DIFFERENCE A B) (ADD C D))
```

can first compute the difference and then the sum, or the other way round.*

* This, of course, is conditioned by the absence of side effects, which is why we use the term 'functional'.

Although functional concurrency is non-parametric, and thus the gain is never as dramatic as one from the use of data concurrency, it is nevertheless important to allow for functional concurrency as an auxiliary mechanism to implement the parallelism of data. It can be used for static scheduling of data flow in a multiprocessor, but also as an additional source of parallelism to hide the latency of remote memory access in a distributed system.

Whereas data concurrency should be introduced by provision of nonscalar operations, functional parallelism manifests itself in the notion of expression. Typical assembly languages do not have this notion, so functional concurrency is only implicitly present in the assembler program. It is therefore desirable to preserve this concurrency in the PSP notation if the PSP is to support data parallelism. This is the reason that the syntax of F-code is made similar to that of Lisp: the F-program is a tree encoded in the form of list. Each vertex of the tree is a function (F-instruction) invocation with its subtrees executing in parallel for most of the vertex labels (or F-instructions).

It should be emphasized that the support of functional concurrency in F-code does *not* mean that F-code is a functional language. A functional PSP would not be able to efficiently support imperative languages because of the principal divergencies in the two programming paradigms.

2.2. Data

Most of the programming languages introduce a concept of type. Being a PSP F-code suggests a consistent generalization of those concepts in application to data parallelism. The F-code convention allows F-instructions to be stripped of unwieldy attribute specifications, at the same time providing for inferrability of those attributes when the platform is to be compiled. In addition, F-code introduces implicit coercions.

All that helps generating compact F-programs from language compilers and minimizes interpretation overheads where the platform is to be interpreted.

2.2.1. Type and sort

F-code types are subdivided into a fixed set of basic types which form a hierarchy in the sense of compatibility:

$$logical < character < integer < real < complex$$

and structural metatypes, or templates, which are user-defined and 'soft' so that any pointer can be dereferenced using any metatype.

Since data-parallel computation normally involves geometric transformations that select and re-order elements of non-scalar objects and since the results of such selections can act as objects in their own right, it is important to provide some sort of address arithmetic. The general mechanism of pointers is softly typed to allow for structures, therefore a non-scalar object can not be represented by a conforming nonscalar aggregate of pointers without discarding its type; even if it could it would not be efficient to dereference this aggregate every time the value is required: array objects are normally stored in memory in some regular way, which implies that the pointer array in question would often be redundant.

In order to solve this problem, every object is ascribed an attribute of *sort*, which can be *value*, *name* or *target*. A target (which is indeed a target for assignment) is a strongly typed object each element of which refers to a value element whose type is equal to, or senior than, the type of the target (thus the compatibility of basic types is taken into account). Note that the type hierarchy for targets is reversed, because whereas one can raise a value from, say, integer to real or any higher type, an object of real type can be assigned values of integer or any junior type. The F-functions that provide access to variables can yield not only values but targets as well, which can be processed geometrically and still remain strongly typed even after they are mixed up with other targets to form the left-hand side of an assignment. Thus F-code promotes a greater functional concurrency by allowing expressions on the left-hand side. Other PSPs for data-parallel processing, such as VSA (Jesshope, 1989) were less neutral architecturally and therefore had to rely upon masked assignment only.

It should be emphasized that the fact that targets refer to values element-wise is an F-code concept. F-code does not assume any specific way the targets are implemented. In particular there is no need to maintain an address table to represent a target if the object the target refers to can be characterized with a simple descriptor.

A target can not be dereferenced since its type is generally incompatible (lower) than the one of the value that would come out as the result, in the sense of type hierarchy of values. There are situations where a target is required not only for assignment, but also for dereferencing (which is the case with read/write variables). In that case a name is used which has a rigid (not coercible) type and which subsumes the functionalities of a value and a target.

The above conventions of sort are part of the F-code paradigm; the important thing is that they constitute a mechanism that makes it redundant to specify access to data objects in each F-instruction. As well as type, the sort attribute is *inferrable* in the F-program, the absolute minimum of access specification is therefore required.

It is worth mentioning finally that architectures introduce attributes of data that are not defined in the architecture-neutral approach, most importantly the mapping function that determines the physical distribution of data among the units of store. Wherever that could be deduced from a (high-level) program it should be, but then F-code does not make such analysis any more difficult. The authors' view is that for a truly generic implementation of F-code a fast run-time solu-

tion should be sought; that however is well outside the topic of this paper.

2.2.2. *Variables and scopes*

The main mechanism of memory scheduling in F-code is a stack, since this is the case in many of the top level languages. Also, stack scheduling in a pseudo-functional platform (which F-code actually is) is natural: evaluation of a program is effected through recursive function invocation, which uses a stack anyway.

Specifically, F-code supports a lazy function which creates a variable that exists during the function activation. One of the arguments of this function (executed after the creation of the variable) is the scope of the variable. Access to the stack variable inside the scope is either through its identifier (similar to Lisp, F-system maintains a dictionary of the active associations of all identifiers occurring in the F-program) or through a pointer to it.

F-code supports the traditional primitives to control heap memory as well.

2.2.3. *Polymorphism and inferrability*

Generally speaking, the majority of F-instructions are polymorphic. For example, F-function DYADIC with operator ADD does integer addition when the operands are integer and real addition when they are real. It also works when the types of the operands differ from each other, in which case it coerces the data with the lower type. A more dramatical example of polymorphism is F-function COMPOSE which glues two objects. This function admits the operands of all sorts and types and works differently for values, names and targets.

Polymorphism drastically reduces the proliferation of F-functions thus facilitating interpretation of F-code. However, if an F-program is to be compiled, polymorphism may present a problem, since the F-compiler has to know for which type data the F-function should execute (in other words, data attributes have to be inferred). The solution suggested by F-code is to allow polymorphism inside an F-module only. Every identifier can therefore be retraced to the outmost environment where it is explicitly associated with some object. The variables that are essentially external parameters of the module do not have such an environment inside the module. To avoid the polymorphism caused by those parameters, there is a special F-function which coerces a variable to some specified attributes.

2.2.4. *Control*

The semantics of F-code is purely operational: there is neither predefined program structure nor restrictions whatsoever as to allowed combinations of instructions. Even the scoping of data is determined by lazy evaluation of F-functions, put another way, by the enforced ordering of argument evaluation for a function. The only way not to break the operational semantics and at the same time avoid an explicit instruction address space is to get the flow of data through the function calls that form the F-program to carry some control tokens which cause decision making at control vertices of the program graph.

In effect, the sole reason for control constructs in an imperative language is to determine the order in which assignments are executed. Since the assignment statement regarded as a function has only a side effect, which is changing the value of the left-hand side, F-code defines its main effect as returning some normal completion code. The completion code (or *token*) is merely an integer scalar value, which may be returned by an 'ordinary' F-function, like the one performing an arithmetic operation. To enforce sequential composition of assignments F-code introduces a set of lazy functions which evaluate their arguments in some order. The action such functions perform on the argument values (completion codes) is to test the tokens and decide whether the execution should be continued; these control functions yield completion codes themselves to allow for nesting of control constructs.

It should be noted that this approach to organization of control assumes only reducible data-flow graphs can be represented by F-code, i.e. of the kind the language compiler generates from fully structured source programs (see Aho *et al.*, 1986). Although this involves some minor difficulties at the stage of code generation from unstructured languages like Fortran, it does, however, facilitate optimization techniques on F-code. Another important fact is that the conversion of F-code branches to a linear representation can always be performed at the F-code compilation stage and therefore does not have to be emulated at run-time.

The overall control model is exemplified in Figure 1. In essence, the virtual F-machine operates in two modes: eager and lazy. The lazy mode is usually triggered by an assignment completion. The eager mode is entered every time a new assignment (or stand-alone expression, e.g. an IF condition) is to be executed. In a way the eager (functionally parallel) evaluation of expressions is

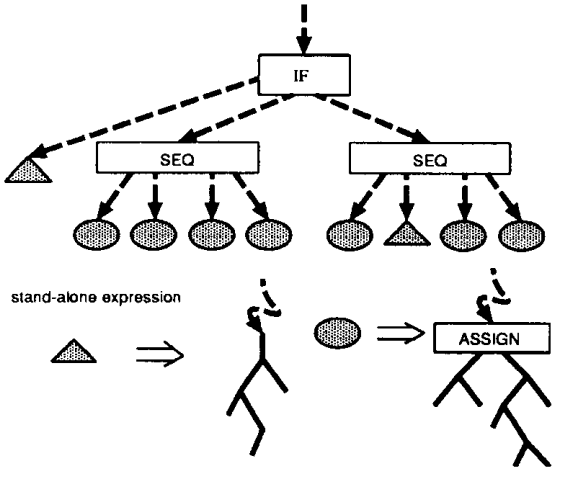

FIGURE 1.

subordinated to the lazy (sequential) execution of the program as it should be. Still, a sequential piece of code can be included in an eager span of the program if necessary (the most obvious example is the C operator ?: that is lazy but computational). This is achieved through a special two-argument lazy F-function which has one token argument, which it fires first and then absorbs, and one object argument that it passes on. Above the invocation locus of this function the evaluation becomes eager.

3. OUTLINE OF F-CODE

This section details some specific features of F-code. It is included in the present paper in order to demonstrate the way F-code is defined as well as to show a few interesting solutions we have found to the problem of accurate representation of data-parallelism in an architecture-neutral manner. This section does not therefore endeavour to give a complete definition of F-code. The interested reader is referred to our internal report (Bolychevsky *et al.*, 1992) which contains such a definition.

3.1. Objects

The F-functions receive as operands, and return as results, *primitive* objects, which are homogeneous, scalar or data-concurrent nonscalar, aggregate of scalar elements having the following characteristics:

1. Type $t \in \{logical, character, integer, real, complex\}$. The types form an ascending hierarchy in the same order as they are listed above. For example, a character type is junior to the real type and senior to the logical type.
2. Sort $s \in \{value, target, name\}$. If an object is of sort value, it contains data of type t; if it is a name, every element of it contains a reference to a value element of type t; if it is a target, every element of it contains either a reference to a value element of type $t' \geqslant t$ or a special dummy reference. Targets are used for element-wise movement of values in the course of assignment. A name can be used the same way as a target, but also admits data-parallel dereferencing, yielding a value of type t.
3. Shape $\mathbf{d} = \{d_i\}$. It is a vector of the extents of the object's dimensions d_i, each of which is a non-negative integer number. The length of **d** is equal to the object's rank. In particular, if the object is scalar, vector **d** has zero length.
4. Contents c, which form an array of shape **d**.

The F-program can manipulate heterogeneous aggregates of data as well, such as Pascal records of C structures. Characteristics of such *structured* objects are as follows:

- *Template* (metatype) T, which is a tree whose leaves are types; all the nodes of this tree except the root one are labelled with shape vectors. Thus, at each level of hierarchy a template determines a sequence of fields. The number of the fields is equal to the number of successors of the node, while the types (or templates) and shape vectors of those fields are defined by the corresponding leaves (or subtrees) and their labels.
- Shape **d**.
- Contents c, which form an array of shape **d** built up from structures of template T.

Structured objects can only consist of values. However, while accessing their homogeneous fields, primitive objects of any sort may arise. Such an access is effected through *pointers*, which are primitive objects of integer type. Pointers are also used for references to subroutines.

3.2. Representation of the F-program

The F-program is represented in a Lisp-like list form as a system of nested lists. Two kinds of atoms are used:

- Literals, which include function labels, keywords (which modify the semantics of F-functions), constants and bit masks, which are used to specify the operand orientation.
- Identifiers, which are the only semantically variable entity of F-code. Identifiers are associated with primitive objects and templates.

3.3. Evaluation of the F-program

The evaluation rules are as follows:

- The first item of the function list is a literal determining the function to be applied to the other items of the list.
- Subsequent items are literals and identifiers the function needs to determine the activity it is to provide, or the arguments of the function, which are expressions that the F-system evaluates prior to the function invocation, unless otherwise stated in the function definition.
- Any F-function that assumes its arguments to be evaluated before its invocation does not require any specific order of that evaluation, only requiring that all its argument objects should be available before it is activated. These are called *a priori* operands. The F-function deals with *a posteriori* operands, which are obtained from the *a priori* ones by applying an optional *type coercion*. The coercion changes the operand type: a value going up the hierarchy generalizing the contents element-wise and a target going down with the contents preserved. A name can not be coerced. The *a posteriori* type of each operand is determined by the semantics of the function. Some operations (for example, addition) can work with operands in a certain range of types $t^J \ldots t^S$ (integer ... complex for this example); in such a case the common *a posteriori* type $\hat{t}$ of all operands is expressed in terms of their *a priori* type $t_1, \ldots, t_n$ as

follows:

$$\hat{t} = \begin{cases} \min(t^S, \max(t^J, t_1, \ldots, t_n)) & \text{for values} \\ \min(t_1, \ldots, t_n) & \text{for names and targets} \end{cases}$$

If the *a priori* type of an operand happens to be different, then the coercion will normally take place. If the coercion is impossible, e.g. if it attempts to lower the type of a value or change the type of a name, this will result in an error.

3.4. Operations

The effects of F-operations are defined by providing formulas which evaluate the characteristics of the result from those of the operands. Here are some auxiliary formulas which we use in spatial juxtaposition of the operands and the result.

- The contents array of a primitive object is indexed with a *multi-index*, which is a vector of nonnegative integers. Let us denote the length of **a** as $r(\mathbf{a})$ and introduce a partial ordering of such vectors. If a vector **a** precedes a vector **b**, it is written $\mathbf{a} \prec \mathbf{b}$, if $r(\mathbf{a}) = r(\mathbf{b}) = r$ and $a_k < b_k$ for all $0 \leqslant k < r$. All the other operations with vectors that we will use below should be interpreted element-wise.
- For a vector **a** of some length and a mask m of the same length we define a *projector* as follows:

$$\mathbf{P}: (\mathbf{a}, m) \to \mathbf{b},$$

where $b_k = a_{x(k,m)}$ and $x(k, m)$ is the number of the k-th unity bit in the mask m. The length of the result vector is equal to the number of unity bits in the mask.
- Let us introduce an *expander*:

$$\mathbf{E}: (\mathbf{a}, m) \to \mathbf{b},$$

so that

$$b_k = \begin{cases} \infty & \text{if } m_k = 0 \\ a_{X(k,m)} & \text{otherwise,} \end{cases}$$

where $X(k, m)$ is the number of mask bits $m_j = 1$ with $j < k$. Expander $\mathbf{E}(a, m)$ is determined only if m has the number of unity bits equal to the length of **a**. The length of **b** is equal to the number of bits in mask m. Thus the following identity takes place:

$$\mathbf{P}(\mathbf{E}(\mathbf{a}, m), m) \equiv \mathbf{a}.$$

In order to refer to a specific operand of the F-function, we use the operand's denotation as a superscript (for example, $\mathbf{d}^x$ is the shape vector of operand x). The following shorthand notation is also needed:

$$\mathbf{p}^x(\mathbf{k}) = \mathbf{P}(\mathbf{k}, m^x),$$

$$\bar{\mathbf{p}}^x(\mathbf{k}) = \mathbf{P}(\mathbf{k}, \overline{m^x}),$$

$$\mathbf{e}^x = \mathbf{E}(\mathbf{d}^x, m^x),$$

where m^x is the mask preceding operand x and $\overline{m^x}$ is the complement of m^x:

$$\overline{m_i^x} = \begin{cases} 0, & m_i^x = 1 \\ 1, & m_i^x = 0. \end{cases}$$

The meaning of this shorthand can be expressed verbally as follows: $\mathbf{p}^x(\mathbf{k})$ is the projection of the coordinate vector **k** on the mask of operand x, $\bar{\mathbf{p}}^x(\mathbf{k})$ is the projection on the complement of this mask, and $\mathbf{e}^x$ is the expansion of the shape vector of the operand x by its own mask.

Let us denote as $s^x(\mathbf{k})$ an array containing all the elements of some layer of the operand x contents. Its shape is $\mathbf{d} = \mathbf{p}^x(\mathbf{d}^x)$ and its elements are determined by the following formula:

$$s^x(\mathbf{k})_\mathbf{l} = c^x_{\min(\mathbf{E}(\mathbf{k}, \overline{m^x}), \mathbf{E}(\mathbf{l}, m^x))}, \quad \mathbf{l} \prec \mathbf{d}.$$

In fact, this formula represents the contents of x as a family of layers, each of which is parallel to the coordinate axes marked by ones in the mask of this operand.

3.4.1. *Data-parallel operations*

F-code supports a vast set of data-parallel operators, which, apart from conventional arithmetic and logical operations, includes basic functions, bitwise operations and shifts. All of them are uniformly generalized to the non-scalar case with the help of F-functions MONADIC and DYADIC.

Function MONADIC has the following form:

```
( monadic <unary> EXPR.a )
```

where <unary> is a keyword denoting a specific unary operator, and EXPR.a is the F-expression that yields operand a. The result of function MONADIC is computed by data-parallel application to a of the operator $\odot$ determined by <unary>:

$$\mathbf{d} = \mathbf{d}^a,$$

$$c_\mathbf{k} = \odot a_\mathbf{k}, \quad \mathbf{k} \prec \mathbf{d}.$$

Function DYADIC uses orientated operands:

```
( dyadic <binary> <mask> EXPR.l <mask> EXPR.r )
```

Its result is computed as follows:

$$\mathbf{d} = \min(\mathbf{e}^l, \mathbf{e}^r),$$

$$c_\mathbf{k} = c^l_{\mathbf{p}^l(\mathbf{k})} \odot c^r_{\mathbf{p}^r(\mathbf{k})}, \quad \mathbf{k} \prec \mathbf{d},$$

where $\odot$ is the operator determined by the keyword <binary>.

The mechanism of orientation allows to freely combine operands of different ranks. For example, to compute a matrix as the tensor product of two vectors, one should specify mask 01 for one of the operands and mask 10 for the other one. Note that the formula for the shape vector of the result coerces the operand extents along coincident axes.

3.4.2. Other computational operations

Function REDUCE ensures both the complete reduction (to a scalar) of the operand, or a partial reduction (along any subset of the axes specified by the mask):

(reduce <total> <mask> EXPR.a)

Here <total> is a keyword which denotes a specific cummutative, associative binary operator. F-code supports only that sort of reductions, since the parallel implementation is assumed. The result is computed as follows:

$$\mathbf{d} = \bar{\mathbf{p}}^a(\mathbf{d}^a),$$

$$c_{\mathbf{k}} = \bigodot_{\mathbf{l} \prec \mathbf{p}^a(\mathbf{d}^a)} s^a(\mathbf{k})_{\mathbf{l}}, \quad \mathbf{k} \prec \mathbf{d}.$$

where $\odot$ is the reduction corresponding to <total>.

The function POL evaluates a polynomial of an arbitrary number of non-scalar variables.

(pol EXPR.c {<mask> EXPR.v}...)

Operand c determines the coefficients of the polynomial, and operands v_i, $0 \leqslant i < n$, determines the values of variables on which to compute it. An equality $n = r(\mathbf{d}^c)$ must be satisfied. The result is computed as follows:

$$\mathbf{d} = \min_{0 \leqslant i < n} \mathbf{e}^{v_i},$$

$$c_{\mathbf{k}} = \sum_{\mathbf{l} \prec \mathbf{d}^c} c^c_{\mathbf{l}} \prod_{0 \leqslant i < n} (c^{v_i}_{\mathbf{p}^{v_i}(\mathbf{k})})^{l_i}, \quad \mathbf{k} \prec \mathbf{d}.$$

Here the mechanism of orientation and shape coercion is the same as in function DYADIC.

3.4.3. Geometry

F-code supports a vast group of geometric operations that 'cut out' fragments of the operand (SECT, SLICE, DIAG, GATHER), 'glue' objects to one another (REPL, COMP), or rearrange the elements of the operand (TRANSP, PACK, TRANSFORM). All of these have the most general data-parallel semantics. As an example, consider function DIAG, which selects main diagonals of all layers of the operands:

(diag <mask> EXPR.a)

To rigorously define its semantics we shall need two additional vectors: $\mathbf{A}(\mathbf{v}, x)$ is vector $\mathbf{v}$ with a component x appended; $\mathbf{R}(\mathbf{v})$ is vector $\mathbf{v}$ with the last component removed. Also denote the last component of vector $\mathbf{v}$ as $L(\mathbf{v})$ so that $\mathbf{A}(\mathbf{R}(\mathbf{v}), L(\mathbf{v})) \equiv \mathbf{v}$. The result of function DIAG is computed as follows:

$$\mathbf{d} = \mathbf{A}\left(\bar{\mathbf{p}}^a(\mathbf{d}^a), \min_{0 \leqslant i < n} p_i^a(\mathbf{d}^a)\right),$$

where $n = r(\mathbf{p}^a(\mathbf{d}^a))$ is the number of unity bits in the mask;

$$c_{\mathbf{k}} = s^a(\mathbf{R}(\mathbf{k}))_{\mathbf{q}(\mathbf{k})}, \quad \mathbf{k} \prec \mathbf{d},$$

where

$$q_i(\mathbf{k}) = L(\mathbf{k}), \quad 0 \leqslant i < n.$$

The most powerful geometric operation is TRANSFORM, which is a non-scalar indexing with the standard spatial juxtaposition of index aggregates:

(transform EXPR.s {<mask> EXPR.t}...)

Operands t_i, $0 \leqslant i < n$, are integer multi-indices. An equality $n = r(\mathbf{d}^s)$ must be satisfied. The result is computed as follows:

$$\mathbf{d} = \min_{0 \leqslant i < n} \mathbf{e}^{t_i},$$

$$c_{\mathbf{k}} = c^s_{\mathbf{l}(\mathbf{k})}, \quad \mathbf{k} \prec \mathbf{d},$$

where

$$l_i(\mathbf{k}) = c^{t_i}_{\mathbf{p}^{t_i}(\mathbf{k})}, \quad 0 \leqslant i < n.$$

3.5. Data management

The basic mechanism of F-code data management is a stack of enclosed scopes. It is implemented with F-function CREATE which works as follows:

1. Create primitive or structured variable as case may be and denote it by identifier. Store old association of this identifier.
2. Compute lazy argument. (It is in fact the new scope for the variable.)
3. Purge variable and restore old association for identifier.
4. Return what was returned by lazy argument.

A primitive variable is created as a couple of objects: a value and a name referring to this value element-wise. The identifier is associated with the whole couple thus providing access through F-function VAR both to the name and the value.

A structured variable is created as a memory pool. The identifier is associated with a pointer to this pool, which is a scalar integer value. Access to a structured variable with F-function SELECT requires a template, a field number and an access mode specified along with the pointer. Depending on the access mode (which is a sort specification) function SELECT yields either the primitive field itself or a name/target referring to this field element-wise. If the required field happens to be structured, then the result of the field selection is a pointer to that field which can be subjected to further selection, etc. The template specified in function SELECT may be different from the one supplied while creating the structured variable. So, it is possible to superimpose different structures onto the same memory pool. A non-scalar pointer, too, may be used in selection, in which case a data-parallel selection takes place and the dimensionality of the result is the sum of those of the pointer and the field selected.

In addition to the stack-based data management, F-code provides tools for using heap memory. These tools resemble the global/local generators of Algol-68 and the malloc/salloc mechanisms of the C language. F-function GLOBAL creates a primitive or structured

variable, but unlike CREATE it yields a pointer to this variable as the result. Such a variable may be purged only explicitly, by F-function DISPOSE. F-function LOCAL is similar to GLOBAL, but it creates the variable in the local heap, which is automatically purged as a whole when the nearest enveloping F-function MARK terminates. Access to heap variables is always using the function SELECT.

Data-parallel address arithmetic of the C style is supported for pointers. An F-function DISPLACE displaces the pointer by the specified number of steps while F-function DISTANCE computes the distance between pointers in steps. Both these functions require the template of the variables pointed out by the operands (in the C language this information constitutes the pointer type).

3.6. Control constructs

The following control F-functions are supplied: SEQ (sequential composition), PAR (parallel composition), LOOP (sequential loop), SPAWN (parallel composition of identical activities parameterized by a number or 'parallel loop') and IF (selection of one of two activities).

All the functions of this group yield integer scalar values treated as completion codes. These codes are analysed after each evaluation of an argument of SEQ or LOOP function. Code zero causes computation to normally continue, a negative code signals that the F-program must be stopped immediately. If a positive code is received then the function SEQ or LOOP terminates and returns this code less one. This enables the completion of any enveloping linear sequence or loop. For correct interpretation of completions the functions PAR and SPAWN (both performing process-parallel computations) yield the maximum of the completion codes of their arguments if all these happen to be non-negative (which causes the farthest completion) or the minimum of them otherwise (which asserts the maximum error code).

4. IMPLEMENTATION

Since F-code is defined rigorously in terms of data-parallel algebra, this provides a rigorous implementation guide, but not a strait-jacket and so it is implementable in any number of ways, one of which will be described here. There are any number of computational modes one can bear in mind before setting out to implement F-code. F-code is a functionally and geometrically parallel platform and favours computational models such as ARAM (Jesshope, 1991) and ETS Data Flow [which is an essentially unrestrained array-based data-flow model (Papadopoulos, 1991)].

The front-end of a compiler consists of four classical phases: (1) lexical analysis, (2) parsing, (3) type inference and (4) shape inference. Since F-code is an intermediate language, it is free of syntactic sugar and its simple Lisp list-like format makes parsing particularly easy. These four steps are also contained in the MOA compiler (Hains *et al.*, 1992) Static shape inference is not new: some APL interpreters infer *forms* (shapes) to minimize space requirements, however its use in implementing lazy evaluation seems to be novel.

4.1. Selective assignment

APL2 includes the notion of selective assignment in a very simple way in which only those parts of a computation which are active have to be evaluated. This is extended in F-code. Figure 2 shows two data parallel assignments. If the darker shaded areas of the right and left hand sides of Figure 2 are all that is actively required of a computation there is no necessity to evaluate any other part. This is especially important for an implementation on a scalar or vector machine, because redundant evaluation affects the execution speed very much. Part of this implementation traces the most selective subset of a computation through a tree of F-code instructions whose semantics effectively combine to produce a lazy evaluation in the spatial (data-parallel) domain. This will be demonstrated after introducing a few more parts of the implementation.

4.2. Type inference

The initial F-code representation of a computation is devoid of all but necessary type information. A type-inference procedure is applied to the parse tree to prove that it can be implemented monomorphically and that it is interprocedurally type-safe. The type inference procedure rewrites the original tree to include explicit type-coercions and if they exist, type errors. If there are type errors the compiler should terminate abruptly since the program is not monomorphic. F-code is an intermediate language which means that it could permissibily assume that a high-level compiler producing code for it produces code which is *type-safe* in which case type errors never

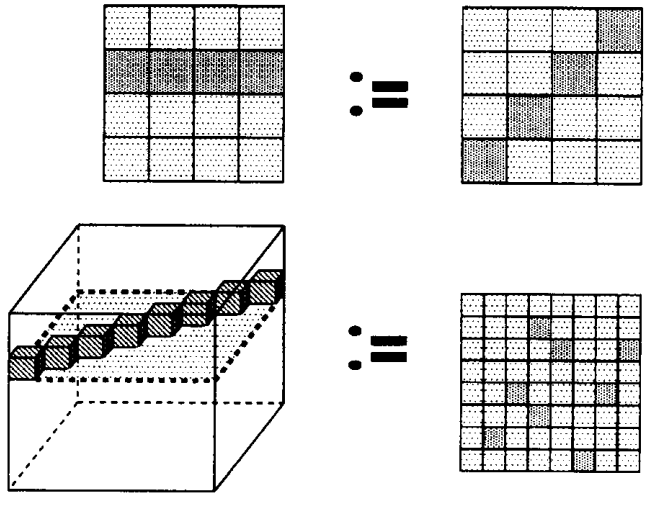

FIGURE 2.

occur; but since it is also an intermediate language for prototype compilers, type errors are still detected.

Type-inference is a swift pattern-matching algorithm which permits the original representation to omit all but necessary type information therefore making the initial format more succinct.

The F-code programs are annotated with the types attributed to F-code operations. This is not part of the definition which needs to be known to targetters to F-code. The type-inference is a localized procedure in most cases which consists of asserting an appropriate type along the three arcs (left, right and up) of a parse-tree node. Type-clashes result in the addition of type-coercion nodes. This is shown in Figure 3 above: the addition of two integer operands results in an integer; the addition of a real and an integer results in a real addition, requiring a coercion of the integer operand. As well as the basic type of the operation, the inference procedure also percolates the rank and sort of nodes up and down the tree simultaneously.

4.3. Shape inference

In order to compile F-code a recursive inference procedure is applied to the parse tree to evaluate expressions for the shape of every operation. F-code is annotated again to demonstrate this part of the procedure. The shapes inferred for F-code expressions are themselves F-code expressions. The shape-inference procedure depends on the shape definitions of F-code's algebra.

For dyadic operations

$$\mathbf{d} = \min(\mathbf{e}^l, \mathbf{e}^r),$$

and shape inference can be demonstrated with a number of examples on dyadic operations.

```
( [(dyadic min X1 X2)] dyadic add
    ( [X1] ... )
    ( [X2] ... )
)
```

Where ... are arbitrary F-code expressions returning vectors and square brackets are the annotation for shape. The shapes of the vectors are X1 and X2 and so the result is the minimum of these. If X1 and X2 are known to be constants at compile-time, the extent expressions may be folded to constants.

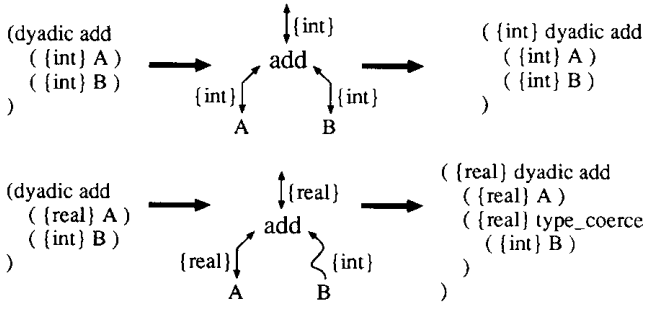

FIGURE 3.

```
( [4] dyadic add
    ( [8] ... )
    ( [4] ... )
)
```

Which says adding a 4 element vector to an 8 element vector produces a 4 element vector.

Shape inference generally inherits the shapes of operands as common sub-expressions.

```
( [(dyadic min (dyadic min X1 X2) X3)] dyadic add
    ( [(dyadic min X1 X2)] dyadic add
        ( [X1) ... )
        ( [X2] ... )
    )
    ( [X3] ...
)
```

Orientation masks are dealt with similarly:

```
( [X1, X2] dyadic mul
10  ( [X1] ... )
01  ( [X2] ... )
```

Produces a matrix result of size X1 × X2 from two vectors of size X1 and X2.

Using the shape rules it is possible to infer the extents of all F-code operations. If certain shapes are not known at compile-time the extent expressions may include variables.

4.4. Selective code generation

In this section an example of generating code in a selective way will be given. The target code is scalar C. It is worth mentioning that this compiles lazy data-parallel algebra to produce eager scalar node programs. The program may be thought of as a node program parameterized by its coordinates in the iteration space of data-parallel computation.

In Figure 4, the program's shapes are inferred resulting in the numbers to the right hand side of the program. The shape inference produces a number of graphs which are in effect dependency graphs between indices. The

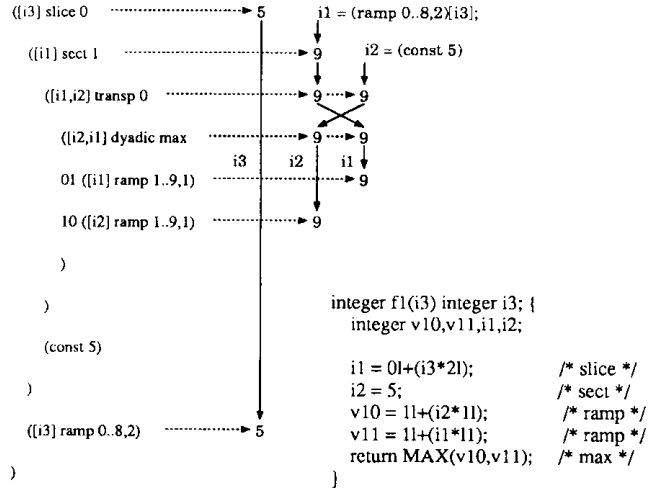

FIGURE 4.

indices are labelled on these graphs (i1 ... i3) and these indices are annotated in the original F-code program. i3 is an incoming index from the environment, i2 is fixed to a constant because it is an internal index created by the sect. i1 is computed from i3 because it is an internal index of the slice.

It is interesting to note the effect of the transpose operation in this graph. The indices are merely swapped around and the transpose takes no time to compute since it is little more than a directive for indices.

The transformation of this referentially transparent section of F-code into its parameterized C equivalent is very straightforward. This program may be executed on five processors, or sequentially on a single processor. The dependences between indices (the graphs produced by shape inference) may also be used for data-distribution in some way although this has yet to be fully cogitated since the first implementation of F-code is to a single i860 system.

The C node program is shown on the right. This program is the most selective implementation of the F-code program. The most selective implementation will always be gained by this method. The most *selective* implementation may not always be the most *efficient* implementation depending upon the instruction set of the target computer. In order to take advantage of, or to avoid parallel hardware features, F-code programs must undergo rewrites.

5. CONCLUSIONS

A general approach to the implementation of data-parallel languages has been presented. The task of producing a data-parallel environment for the end-user is proposed to be structured into two fundamental stages. Firstly a programming language suitable for particular computational needs should be chosen, which could be a data-parallel extension of a sequential language, or a consistently data-parallel one such as EVAL developed by Muchnick and Shafarenko (1994) and recently re-targetted towards F-code by A. Bolychevsky.

Then a compiler for this language(s) needs to be developed to produce F-code as output. The key issue here is that F-code is defined formally which makes it possible to ensure the adequacy of such an output for a given language, and which makes code generation truly machine independent. No optimization is necessary at this stage, therefore, a satisfactory variety of languages can be implemented at once.

The last stage is to develop an F-code environment on all the parallel computers the user-end compilers are to appear. As this paper shows, that is not really a design-from-scratch exercise as parts of the existing F-environments should be re-used with little or no change.

An F-code to scalar C compiler has been written and tested which implements the full functionality of F-code. An F-code to i860 implementation has almost been completed. A full description of the implementation given briefly in this paper will shortly appear in Sutton (1993). We are currently working towards the next, now distributed, implementation of F-code on a transporter network.

REFERENCES

Aho, A. V., Sethi, R. and Ullman, J. D. (1986) *Compilers.* Addison-Wesley, Reading, MA.

Bird, R. S. (1987) An introduction to the theory of lists. In Broy, M. (ed.), *Logic of Programming and Calculi of Discrete Design*, pp. 3–42. Springer-Verlag, Berlin.

Bird, R. S. (1989) Algebraic identities for program calculation. *Comp. J.*, **32**, 122–126.

Bolychevsky, A. B., Muchnick, V. B. and Shafarenko, A. V. (1992) *Functional Representation of a Data-concurrent Program.* Internal report, Department of Electronic and Electrical Engineering, Surrey University, Guildford, UK. This is a complete definition of F-code.

Brown, J. A., Pakin, S. and Polivka, R. P. (1988). *APL2 at a Glance.* Prentice Hall, Englewood Cliffs, NJ.

Chen, M. and Cowie, J. (1992) Prototyping Fortran-90 compilers for massively parallel machines. *SIGPLAN 92 Conf. on Programming Language Design and Implementation. Sigplan Notices*, **27**(7).

DRA (1991) *TDF Specification.* Defence Research Agency, RSRE, Malvern, UK.

Fortran-90 (1991) *Fortran 90 Standard.* Publisher?

Fox, G., Hiranandani, S., Kennedy, K., *et al.* (1992) *Fortran D Language Specification.*

Hains, G., Biberstein, O. and Foisy, C. (1992) *An MIMD Compiler for Functional Programs on Arrays.* Publication 830, DIRO, Département d'Informatique et de Recherche Opérationnelle, Université de Montréal.

Hains, G. and Mullin, L. M. R. (1993) Parallel functional programming with arrays. *Comp. J.*, **36**, 236–245.

HPF (1992) *DRAFT High Performance Fortran Language Specification (Version 0.4).* High Performance Fortran Forum.

Iverson, K. E. (1962) *A Programming Language.* John Wiley, New York.

Jesshope, C. R. (1989) The definition and implementation of an active data model. In Plander, I. (ed.), *Artificial Intelligence and Information-Control Systems of Robots-89.* Elsevier, Amsterdam.

Jesshope, C. R. (1991) Virtual shared memory for the ARAM model of compilation using the MP1 packet routing chip. In *Parallel Digital Processors*, pp. 55–59. IEEE, New York.

Lake, T. and Sloman, B. (1992) *TDF Vectorization: Towards Open Parallel Systems.* GLOSSA, Reading.

Marino, G. and Succi, G. (1989) Data structures for parallel execution of parallel languages. In Odijk, E., Rem, M. and Syre, J. C. (eds.), *PARLE '89 Parallel Architectures and Languages Europe (Lecture Notes in Computer Science. 365/366).* Springer, Berlin.

McCarthy, J. (1960) Recursive functions of symbolic expressions and their computation by machine. *Commun. ACM.*, **3**, 184–195.

Muchnick, V. B. and Shafarenko, A. V. (1994) Data-parallel computing: the language dimension. Monograph to be published.

Papadopoulos, G. M. (1991) *Implementation of a General Purpose Dataflow Microprocessor (Research Monographs in Parallel and Distributed Computing).* Pitman, London.

PSRC (1991) *MIMDizer User's Guide (Version 7.02).* Pacific Sierra Research Corporation, Placerville, CA.

Skillicorn, D. B. (1991) Models for practical parallel computation. *Int. J. Parallel Programming*, **20**, 133–158.

Sutton, C. (1993) *The Implementation of a Portable Software Platform*, PhD thesis, University of Surrey, Guildford, Surrey, UK.

Zima, H., Bast, H. J. and Gerndt, M. (1988) SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, **6**, 1–18.

Zima, H., Brezany, P., Chapman, B., Mehrota, P. and Schwald, A. (1992) *Vienna Fortran—A Language Specification (Vesion 1.1)*. Technical Report, ICASE.