8-1-1980

Hierarchical Analysis of a Distributed Evaluator

Robert M. Keller
Harvey Mudd College

Gary Lindstrom
University of Utah

HIERARCHICAL ANALYSIS OF A DISTRIBUTED EVALUATOR

Robert M. Keller

Gary Lindstrom

Department of Computer Science
University of Utah
Salt Lake City, Utah 84112

ABSTRACT

We outline the analysis of a distributed evaluator for an applicative language FGL (Function Graph Language). Our goal is to show that the least fixed point semantics of FGL are faithfully implemented by the hardware evaluator envisioned in the Applicative Multi-Processor System AMPS. Included in the analysis are a formalization of demand-driven computation, the introduction of an intermediate graphic language IGL to aid in our proofs, and discussion of pragmatic issues involved in the AMPS machine language design.

INTRODUCTION

Programming languages for distributed computing systems are receiving increased attention currently, as are languages based on function application. Distributed systems are of interest because of a desire to exploit potential concurrency in programs. Applicative languages tend to reveal potential concurrency by eliminating arbitrary sequencing within program representations, and by circumscribing side-effects. In addition, applicative languages often allow programs to be written so that their text closely resembles that of a correctness specification, thereby easing verification.

Although the idea of using applicative languages as a basis for concurrent programming has come into vogue only recently, the reader should refer to the prophetic paper [1] for an anticipation of many of the relevant ideas currently being put forth. Subsequent proposals, which share some aspects of our own, include [2] through [7].

Sketched herein is an analysis (i.e. an informal correctness proof) of an evaluator for an applicative language suitable for exploiting the features of a distributed computing system. This evaluator has been proposed for use in the Applicative Multi-Processing System AMPS [8]. Such a proof would be of interest for several reasons:

1. The evaluator has been implemented ([9]), so it is desirable to certify its correctness.

2. Although parts of similar proofs have been sketched, notably in [10] and [11], these proofs have been for serial evaluators, and are for models having fewer machine-level details than the one presented here.

3. A graphical approach to semantics seems to us to be quite enlightening in comparison to the one-dimensional representations largely used heretofore.

4. We intend the present exposition as the first step toward a more comprehensive proof which also involves a storage manager.

The extrinsic language, called FGL (for Function Graph Language), includes features deemed relevant to highly concurrent distributed evaluation. The hardware implementation we consider includes a demand-driven data-flow evaluator for effective support of the data structuring primitives of our language. The implementation naturally provides for single evaluation of common subexpressions and parameters.

Locality considerations give rise to a two-level evaluation strategy for the machine language (ML): at the intra-processor level, a rather rigid structure is imposed, in which each atomic function is executed with bounded value fan-out and communication delay for greatest efficiency. At the inter-processor level it is infeasible to place such a bound, as one function may well have to send its result to others, the number and locations of which are not determinable a priori.

Block storage allocation is used in ML for the following reasons:

1. It enforces locality of communication among nodes which are logically closely related.

2. It permits economical use of address bits by requiring only relative addresses within a block.

3. It avoids the need for code relocation and extensive dynamic binding.

4. Tuples of data values are stored as blocks, or pieces of blocks, permitting fast indexing.

5. Fewer interactions with the storage allocator are required.

6. Blocks may be transmitted and initialized in a "burst mode" of communication, rather than in a word-by-word mode.

The proof that the distributed evaluator is correct with respect to FGL's fixed-point semantics is complicated by the two-level

CH1569-3/80/0000-0299$00.75

block-oriented strategy. For this reason, we have found it convenient to introduce a language IGL intermediate between the extrinsic language and that of the target machine. This language allows the analysis to be naturally decomposed into two levels (not corresponding to the levels of evaluation), but does not appear explicitly in the implementation.

We express the notion of demand and value flow in IGL programs as a state-transition system (cf. [12]). The states are marked IGL graphs, with transitions expressed by a set of formal rules. This system is the basis for the FGL evaluator. The notion of the correctness of such an evaluator with respect to FGL semantics is presented. We then discuss the proof of correctness of the IGL evaluator with respect to ML.

The following diagram summarizes the levels of the hierarchy and their functions.

Acronym	Name	Purpose
FGL	Function Graph Language	Programming
IGL	Intermediate Graph Language	Internal program representation for FGL
ML	Machine Language	Physical program execution

The analysis may be outlined as follows:

1. IGL-->FGL mapping theorem: IGL defines the correct result for FGL.

2. IGL partial correctness theorem: IGL can produce the correct result.

3. ML-->IGL mapping theorem: ML defines the correct result for IGL.

4. IGL finite delay: ML provides a finite delay property for IGL, so that "can" above becomes "will".

5. Pragmatic aspects: Certain invariants desired for implementation reasons hold for ML executions.

FUNCTION GRAPH LANGUAGE

Our extrinsic language, FGL (Function Graph Language) is Lisp-based [13], extended to include non-strict atomic and programmer-defined functions. This permits ease in dealing semantically and pragmatically with unbounded data structures, as discussed in [6] and elsewhere. The components of such structures may be distributed among physical processing elements and concurrently constructed and transmuted, using stream-like communication between computing modules which are both physically and logically distributed. Because of the functional nature of FGL, logical aspects of the computation are insensitive to delay in and among physical elements.

The objects supported are not restricted to streams of simple components, such as characters or records, but also permit components which are functions, other streams, and generally arbitrary data objects. FGL allows treatment of functional objects with full lambda-calculus generality [14].

The **cons** operator of FGL permits an arbitrary number of arguments, thus providing an efficient and natural array capability. The usual **car**, **cdr** selectors are generalized to an indexing selector **select**. For simplicity, however, we will primarily use **car** and **cdr** here; **car** selects the first component of a tuple and **cdr** selects the last. Other aspects of our generalization are discussed in [15].

In this presentation, the set of data objects of FGL will be

Objects = Atoms U Tuples U Graphs U {error} U {?}

where

1. Atoms = Integers U Characters U {NIL}, where Integers is the set of integers and Characters is the set of characters of some alphabet. We assume that NIL plays the role of the Boolean value **false**. Any atom other than NIL and error may play the role of the Boolean value **true**.

2. Tuples: A tuple is a sequence of N Objects, for N an arbitrary natural number.

   The limit of a sequence (i.e. "tree") of nested tuples of objects, as nesting occurs ad infinitum, is an object. For example, the stream of odd prime numbers could be represented as

   (3, (5, (7, (11, (13, ...)))))

3. Graphs: We allow the enveloping of a graph, as described in [16], and its use as a function data object (i.e. as a "closure").

4. error: an error value which propagates itself through each function which demands it as an argument.

5. ?: the undefined object, i.e. the result of a computation which has not yet (and might never) produce any value.

A fully operational system might include side-effect operators, but we prefer introducing them within the context of an applicative style, in which the programmer is highly aware of their use (i.e. their use will be permitted only on tuples which are created as explicitly modifiable). Side-effect operators are not included in the model presented here, with the exception of **read** and **print**, which are described subsequently.

For the purposes of this exposition, a program in FGL appears as either a "function graph" or as a "set of equations" [22]. Each equation is determined by naming a FUNCTION being defined, which has zero or more formal parameters. The function name is equated to the RESULT expression, which involves names of defined functions, names of atomic operators, formal parameters, and imported values. Abbreviations of multiply-used values are provided by LET expressions, which are also equations equating the left-hand side identifier of a BE to the right-hand side expression. The latter expression may involve the identifier on its own left-hand side, as can the function being defined involve itself. Finally, an IMPORTS declaration allows values defined externally to a function definition to be used inside the definition. Algol-like lexical scoping is used, except that imported values are declared implicitly.

When a value defined in a LET.... BE.... involves itself, or when a function f defined in terms of a formal variable x involves the expression f(x), or when a value is defined in terms of an expression which involves the importation of the value itself, we say that there is an "applicative loop". Such loops permit implementation of data structures in terms of themselves, thereby providing for the generation of infinite data structures without either the obvious infinite recursion or use of side-effect operators such as Lisp's **rplaca**. The latter often have the effect of destroying local determinacy, a property useful in verifying concurrent programs.

As an example of a textual representation of an FGL program, consider the following:

```
FUNCTION oddprimes(limit)
LET primes be
        cons(3, primesfrom(5))
RESULT  primes
WHERE

FUNCTION primesfrom(n)
IMPORTS (primes, limit)
LET rest BE primesfrom(n+2)
RESULT  if n > limit
             then nil()
             else if relprime(primes)
                  then cons(n, rest)
                  else rest
WHERE

FUNCTION relprime(stream)
IMPORTS n
LET first BE car(stream)
RESULT  (square(first) > n
        or ((not divides(first, n)
                and relprime(cdr(stream)))
```

The program above generates the list of prime numbers beginning with 3 and not exceeding the value of the argument **limit**. It does so by forming a sequence of numbers, a number being included in the sequence only if it is prime. The primality of the number is tested by using lesser members in the sequence as trial divisors.

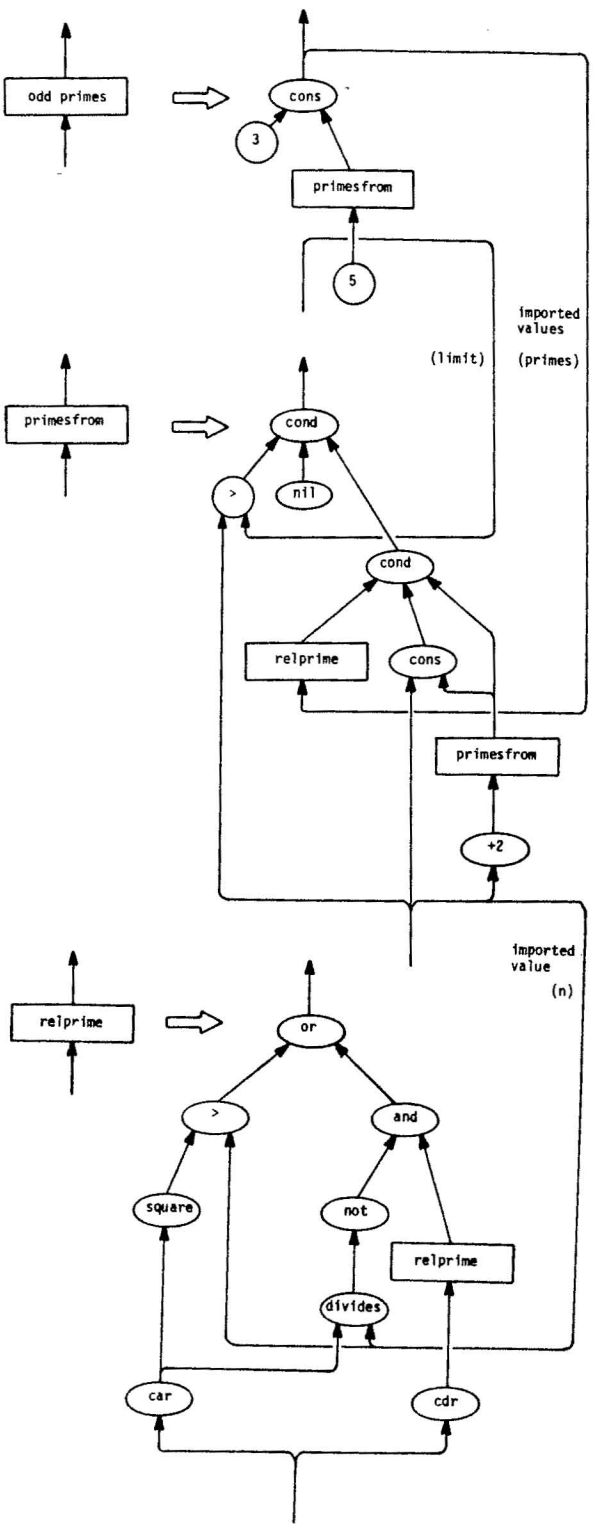

Figure 1: FGL graph of the Odd-Primes Example.

An applicative loop exists, in that primesfrom is used to define the sequence primes, but also uses that sequence as an imported value in its definition. The or above is a sequential function, in that it only demands arguments in sequence as they are needed to determine the value.

An expression in FGL is formally represented as a directed graph, with the nodes being identified with the operators in the expression. We think of each arc in the graph as being a carrier for an FGL data object. A node defines an input/output functional relationship between the ultimate values on the arcs directed into the node and the ultimate value on the arc directed out. (We assume that each node has a single outgoing arc for simplicity.) In the graphical form of FGL, each functional equation may be represented by a graph grammar production in which the antecedent names the function being defined, and the consequent presents the graph of the defining expression.

The graphical form of the preceding program is shown in Figure 1. The applicative loop which results from the compilation of the textual FGL program is evident there. Although in this figure we represent imported values by direct connections into the consequents of productions, accurate treatment of scoping rules demands that productions involving imports be replaced with the concept of an enveloped graph, which may eventually be presented as an argument to the apply function [16]. To simplify the discussion, we shall not consider this treatment here.

Certain atomic functions are provided, such as the following:

add, and, divides, mult, etc. which have the obvious interpretation,

cons groups its arguments into a tuple, even if the arguments are not completely known at time of application. That is,

$$cons(x_1, x_2, \ldots, x_n) = (x_1, x_2, \ldots, x_n)$$

where the right hand tuple exists independent of what the x's might be.

select is defined by

$$select(i, (x_1, x_2, \ldots, x_n)) = x_i$$

provided $i \neq ?$. It is undefined if $i = ?$, but when $i \neq ?$, there is no requirement that $x_j \neq ?$, for any j.

car and cdr are defined by

$$car\ (x_1, x_2, \ldots, x_n) = x_1$$

$$cdr\ (x_1, x_2, \ldots, x_n) = x_n$$

which is consistent with the Lisp definition when $n = 2$.

cond is the name of the conditional function (i.e. "if.... then.... else....").

nil, returns the value NIL.

null, tests for the atom NIL.

Additionally, there are "pseudo-functions", such as print, which has the side-effect of printing its argument on some external device, and read which has the side-effect of reading an external device to determine its result. The use of such functions can be completely avoided outside of utility routines provided for input and output.

Additional auxiliary functions are provided for extra evaluation control. Examples are seq, which causes its arguments to be evaluated in sequence, and par which causes its arguments to be evaluated concurrently. (Strict functions such as add, mult, etc. also have the latter effect.)

Through the use of pre-compilation and removal of certain recursions and common subexpressions, our evaluator incurs no combinatorial explosion of the type which would normally occur in circular recursive evaluation of applicative loops. All theorems proved in [10] also hold for the FGL evaluator. However, the fact that we compile applicative loops without additional recursions provides a feature for yielding terminating executions for evaluations which would be non-terminating in other systems. For example, we can state

Theorem: The FGL evaluator terminates on some programs for which the evaluator of [10] fails to terminate.

To prove this, consider the program (which would differ syntactically when presented to the Friedman and Wise evaluator):

```
FUNCTION main
RESULT  print f(0)
WHERE
        FUNCTION f(x)
        RESULT car f(x)
```

The Friedman and Wise evaluator would recurse infinitely, generating

print(car(car(car(car(...)))))

The FGL evaluator stops (without printing anything) when it dynamically and implicitly "discovers" that f(x) is trying to compute a strict function of itself.

We do not present the fixed point semantics of FGL here, instead referring the reader to [16]. However, we give a brief intuitive description of these semantics. For a directed acyclic function graph, the meaning can be understood simply from the definitions of the functions assigned to each node. That is, the output of each node is the function prescribed for the node applied to the input values of that node. Note that this makes sense even if the graph is infinite, so long as each path from each of the graph's inputs to its output is finite.

In FGL, the program representations are always finite, but these representations can be

understood by (but are not implemented by) expanding the representations into acyclic graphs which are sometimes infinite. Namely,

1. Each node having a function prescribed by a production is effectively the same as replacing that node with the consequent of the production.

2. Each cycle in the graph can be "unwound" by repeated "node-splitting" to obtain an equivalent infinite acyclic graph.

The validity of this means of understanding depends on the fact that all FGL functions are "continuous" over an appropriate Scott data-type ordering. Although this fact is used later, space does not permit further elaboration of its meaning, and the essential ideas may be understood without it. The reader may refer to [16] for further explanation.

Space limitations also preclude further definition of "node-spliting", but the idea is reasonably intuitive. Further discussion may be found in [16]. We henceforth understand by the phrase dag form of an FGL program the acyclic graph as determined above. The above description is equivalent to the "least fixed point" semantics of FGL programs, which is also equivalent to the viewpoint of the program as a system of equations. It also points out the determinacy of FGL programs, i.e. that each program represents a unique function.

The diagram of Figure 2 illustrates the scheme of evaluation in the odd primes example of Figure 1. It shows the loop formed by using the sequence of primes being generated to assist in their own further generation, as well as concurrent evaluation of **primesfrom** for different arguments. The dag form resulting from unwinding the cycle is shown in Figure 3.

Implicitly included in an evaluation such as the one above is an arbitrary number of "producer-consumer" relationships which the evaluator must implement so that needed values are produced and used consistently, independent of system-wide interleaving. These evaluations could be distributed among processing elements to heighten concurrency and thereby reduce computing time. The arbitrary fan-out of values, alluded to earlier, is quite apparent in the diagram.

It should be noted that the definition of FGL semantics is embodied in the language, not the evaluator. That is, its semantics are given denotationally, by specifying the semantics of each of the atomic functions. This is why we prefer to use the term "lenient **cons**" instead of saying that we have a "lazy evaluator" [11]. For a denotationally-defined language, an evaluator is either correct or is not. Similarly, if one wishes a **cons** to have a different effect, this amounts to a redefinition of **cons**, not a change in the evaluator. We happen to prefer the lenient version of **cons** as a standard, but our results in no way rely on the presence of this operator. We can also include other forms of cons (with different names, of course). The main

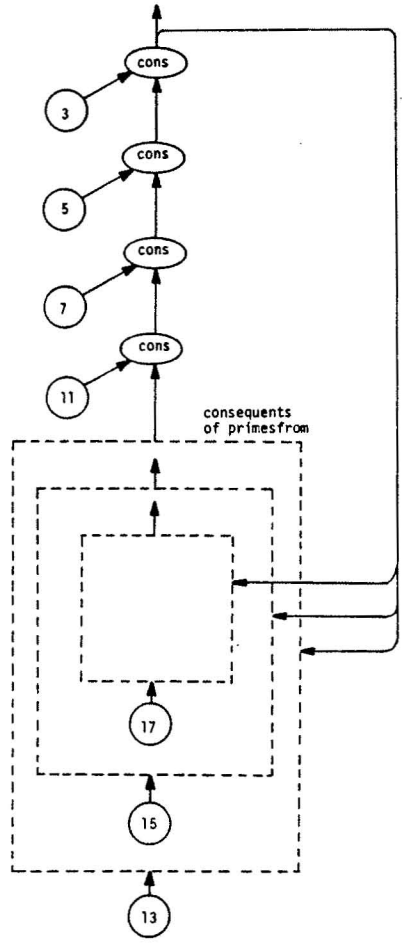

Figure 2: Expansion of the Odd-Primes Example.

reason lenient **cons** is of interest here is because it is the source of a need for "forward chaining", to be discussed later.

A single equation, which defines the "top level" function **main**, acts to drive the others, its value being demanded externally by the system. In a sense, it is the goal of the evaluator to produce the "value" of **main**. For example, we might include the definition of **oddprimes** above in the following program, which reads a number, then prints all odd primes not greater than that number.

```
FUNCION main
RESULT  printall(oddprimes(read()))
WHERE

FUNCTION printall(x)
RESULT   if null x
                then nil()
                else seq(print car x,
                         printall cdr x)
```

The program above, when run on our evaluator, will not produce a particularly high degree of concurrency. However, it is a simple matter to enhance its concurrency with the special operator, **par**, which is functionally transparent (it is the identity function on its first argument), but which has the effect of introducing additional demands for values. In the present example, we need only modify the definition of **primesfrom**, obtaining the following:

```
FUNCTION primesfrom(n)
IMPORTS (primes, limit)
LET rest BE primesfrom(n+2)
RESULT  if n > limit
            then nil()
            else par(
                        if relprime(primes)
                            then cons(n, rest)
                            else rest,
                        rest
                        )
```

In this example, the sub-expression **primesfrom(n+2)** is demanded concurrently with the testing of **relprime(n, primes)**, so that the latter does not cause the generation of the sequence of primes to be sequentialized. Since common sub-expressions are identified as the same value, the same value of **primesfrom(n+2)** will be used in evaluating the if.... then.... else.... No recomputation will take place.

TARGET MACHINE LANGUAGE

As mentioned previously, our ultimate motivation for the FGL evaluator is its realization on the highly parallel machine architecture AMPS. While the physical details of such a machine are not relevant here, its language ML and execution semantics are. Hence we include here a brief sketch of these aspects.

The machine consists of a large number of identical processing elements (PEs), each possessing a portion of a uniformly-addressed, but physically distributed, memory. The fundamental observable action in a PE is a task, involving bounded space and time behavior, such as the execution of an atomic function or the propagation of a value instance or a demand. Parallelism is achieved by exporting, to neighboring processors, function application tasks which have been spawned by strict operators. Unlike the proposal of [7], no "sergeant" tasks are generated for computations which might not be required. However, the programmer may include functions, such as **par** in the preceding example, which cause such tasks to be generated. Further aspects of resource control in FGL are discussed in [17].

Unlike FGL, not every interconnection of ML operators is a valid program. For example, it is possible to construct incorrect linkages. However, the compiler insures that only valid ML programs are generated from their FGL inputs. We have insufficient space to include a presentation of what is or is not valid in ML.

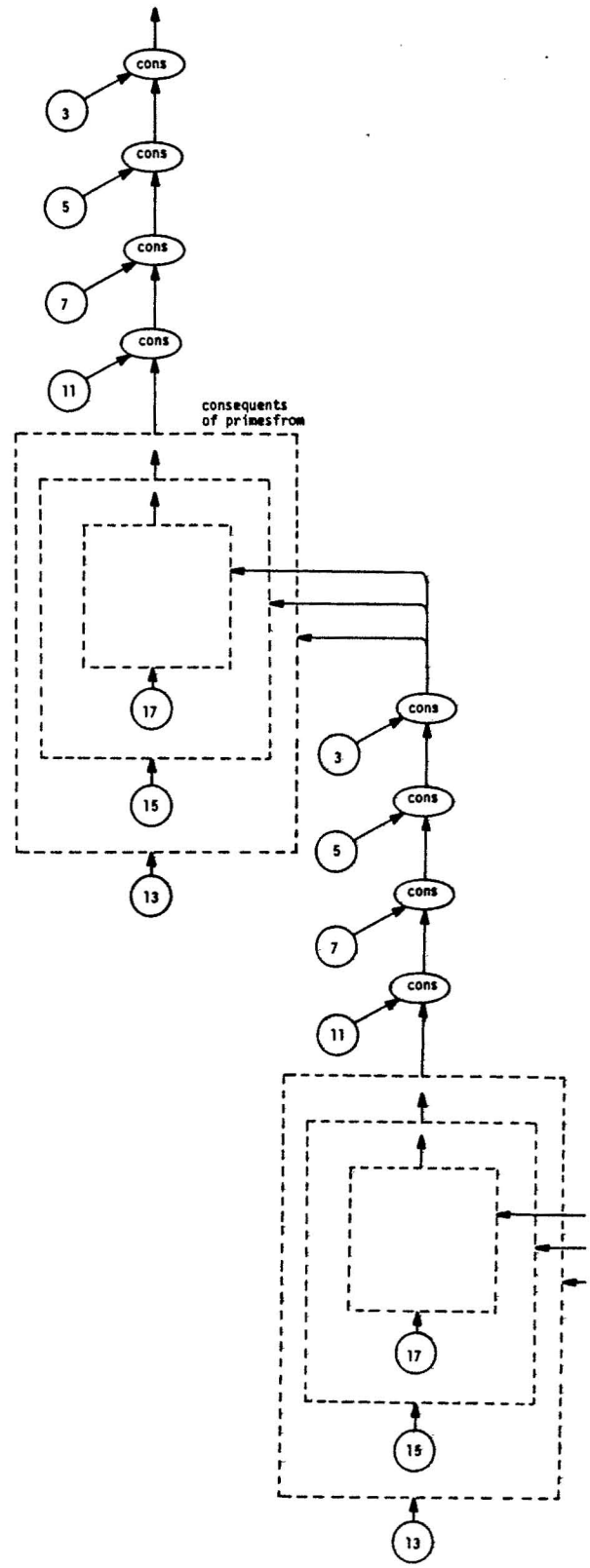

Figure 3: Dag form of the example in Figure 2.

Each programmer-defined function is represented (in pure code) as a block encoding of its graph. The code inside a block has roughly one word corresponding to each node. A typical code word contains the name of the node's operator, local (relative) addresses representing the node's arguments, and space for local notifiers (addresses used to tell which other nodes are to be informed when the node's value is ready).

The action corresponding to application of an FGL production is triggered as each instance of the antecedent is demanded. This action entails the allocation of a block into which the encoded graph is copied, and the linking the arguments and imports of that block with the block containing the antecedent, in effect splicing the graph represented by the code in place of the antecedent itself.

Evaluation of a node entails overlaying the node with its result. Of course, its notifiers are first temporarily saved by the processor, as they occupy some of the space required by the result itself. Here we see a contrast in that FGL values are viewed as appearing on the arcs, whereas ML values appear as transformed nodes. A more important contrast is that FGL objects can be infinite, whereas ML objects must each fit into bounded space.

With these considerations in mind, the FGL model must be refined toward the target machine representation so that fixed word and block sizes are possible. In particular:

1. Because of their disparity in size and meaning, local addresses cannot be freely converted to global addresses and vice-versa. Instead, special operators are provided at compile time to interface from one block to another.

2. While arcs within a block have statically bounded fan-out, global arcs can experience unbounded fan-out (e.g. due to multiple remote demands on a given tuple component's value).

3. In distributing values according to (2), the ML evaluator should not create new nodes (words) to mediate dynamic fan-out, lest storage management become more complicated.

4. The ML evaluator evaluates tasks using a task list which is generally distributed over the available processing elements. This list is used to determine an ordering of the application of transitions. Not all properties of the ordering are important. It only matters that once a transition rule is eligible for application, it does eventually get applied. This effect is achieved by FIFO queuing in ML, and finite-delay is the corresponding property in IGL.

As remarked above, ML code blocks use small relative addresses to express the local connectivity within a function graph. Global addresses are used to represent objects referenceable across code block boundaries. These include references to function definitions (pure code), function closures, function applications (for passage of parameters, globals, and result), and tuple values. In ML execution diagrams, e.g. Figure 7 global addresses will be represented by arcs with hyphenated lines.

The principal operators involving global addresses are **forward** and **fetch**. The operator **forward** connects a local argument (e.g. a function result value) to a global demander (e.g. its place of application). The operator **fetch** does the complementary action. It may be noted in each step that global address arcs only emanate from **forward** nodes, and that no new nodes are created in any step. Thus the creation of global pointers and the use of existing code space is well-disciplined.

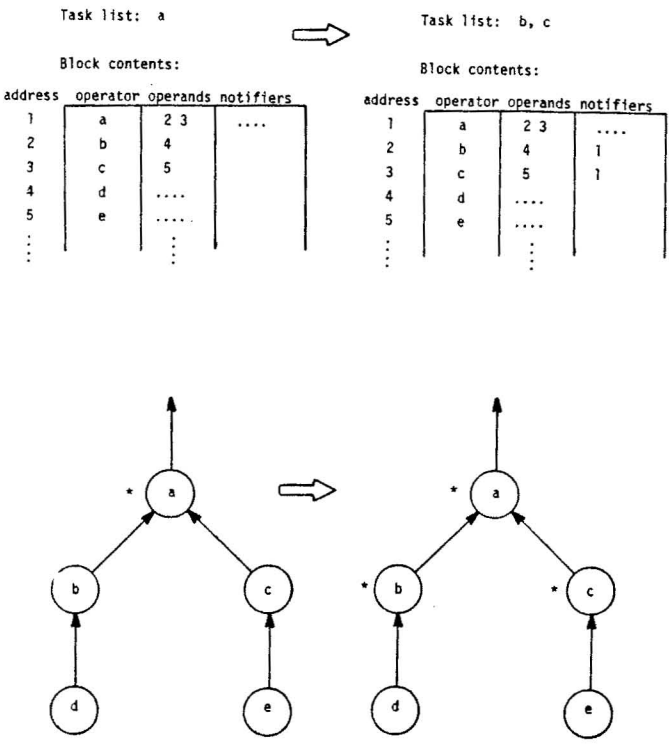

Figure 4: Example of execution in ML and the corresponding IGL transition.

INTERMEDIATE GRAPHICAL LANGUAGE

In attempting to prove that ML is a valid implementation of FGL, the disparity between the two languages seems best approached by the introduction of a third graphical language, IGL. The data objects of IGL are close to those of FGL, except that they use references, whereas FGL avoids references in favor of objects with more mathematical elegance.

The IGL objects are:

1. atoms, as in FGL
2. ?, the undefined value
3. error, the error value
4. references, of one of two kinds:
   a. tuprefs, references to tuples
   b. coderefs, references to master copies of code blocks
   c. funrefs, references to function closures (i.e. pairs consisting of a coderef and a tuple of imported values)
5. tuples of IGL objects of the above types only. (Tuples with tuples as components are not allowed. These must be provided by references.)

Unlike FGL, IGL objects must be finite. There are no limit objects. Instead, limit objects are implicitly represented by fixed points of equations, as will be described presently.

ML and IGL have the same data objects in common, but ML is more restricted in the way it can handle those objects, and includes the special linkage operators mentioned in the previous section. Another common characteristic between ML and IGL is that both are viewed as replacing the operator nodes with a value, whereas FGL is viewed as producing a value on an arc. Hence, we introduce the intermediate language to provide a convenient link between a very mathematical language on the one hand and a very pragmatic language on the other. Table I summarizes the differences between FGL, IGL, and ML. Like FGL, each arc in an IGL graph determines (again, by fixed point semantics) a data object. However, we need to progress toward the operationally defined ML. Hence we must at this point give an alternate, operational, definition of IGL which relates to its denotational definition in an obvious way. Accordingly, we choose to think of the nodes of an IGL graph as having values, which are identifiable as the same values determined on their (single) output arcs. Operationally, an IGL node will ultimately be replaced with that value if there is a demand for it. Another way of viewing this replacement is that the function in the IGL node is changed to a constant function having that value.

The precise operational behavior of our IGL evaluator, as well as its correctness with respect to the denotational semantics, will be approached in terms of "marked IGL graphs", which reflect demand and data flow in a manner similar to ML.

A marked IGL graph is an IGL graph in which each node is either marked *, for demanded or unmarked. Marked IGL graphs are the states of an abstract state transition system (cf. [12]) which models the flow of demand and values among nodes. The transitions in this system are based on transition rules for each of the node operators, as determined by the type of these operators.

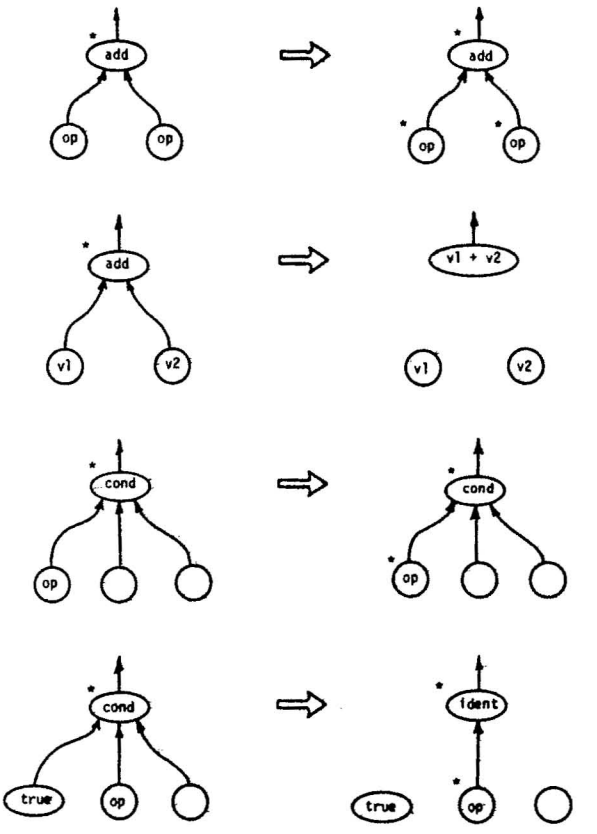

Figure 5: Transitions between marked IGL graphs.

	FGL	IGL	ML
Values manifest	on arcs	on arcs, or replacing nodes	replacing nodes
Infinite values	allowed	not allowed	not allowed
Fan-out	arbitrary	bounded	bounded
Linkages	implicit in productions	tuples and selectors	tuples and selectors converted to fetches and forwards

Table I: Comparison of the three language levels

We list in Figure 5 some of the rules in terms of markings. An evaluator becomes completely specified when the transition rules are accompanied by a specific order for their application. However, in a distributed system, this order will be difficult to control. Thus, instead of giving a rigid order, we assume for IGL only a finite-delay property: A rule cannot remain applicable forever without being applied by the evaluator. This property is insured by the ML realization, as will be later sketched.

IGL TO FGL MAPPING

The use of IGL as a conceptual "implementation" of FGL is achieved through the mathematical device of a mapping from the data values and operators of IGL to those of FGL. As mentioned previously, the main distinction to be drawn between FGL and IGL lies in the data types. Whereas the FGL data types are based purely on mathematical structure, IGL introduces objects which refer to parts of the graph to aid in the progression toward ML.

Another distinction between FGL and IGL of a more technical nature is that the arguments and imports to function objects in FGL are achieved simply by splicing the appropriate arcs together. In IGL, this effect is created by packaging into separate tuples the arguments and imports. These tuples reside in the applying block and the environment block, respectively. Selectors are used inside the applied block which accesses the tuples.

We have already discussed how a unique FGL object is determined on each arc of an FGL program, given that each of its input arcs have been assigned values. In the context of such an input assignment, if x is an arc, then we denote the determined value by Fval(x). In a similar way, a unique IGL object is determined on each arc of an IGL program, and we denote this value by Ival(x).

The IGL program graph gives rise to a system of FGL equations whose least fixed point defines, for each IGL object x, a corresponding FGL object h(x) as follows:

1. If x is ?, error, or atom then h(x) = x.

2. If x is a tupref, referring to

$$(x_1, x_2, \ldots, x_n),$$

then $h(x) = \text{cons}(h(x_1), h(x_2), \ldots, h(x_n))$.

3. If x is a funref, then h(x) is the function graph referenced by x, together with bound import arcs as determined by the tuple part of the referenced closure.

Each arc of the FGL graph can be identified as a unique arc of the IGL graph. Since IGL has additional operators for linkage, the converse is not true.

The link between the partial correctness of IGL and that of FGL may now be stated in terms of an equation involving the mapping h.

IGL-->FGL mapping theorem: For any arc x of an FGL graph,

$$\text{Fval}(x) = h(\text{Ival}(x))$$

To prove this theorem, we need only observe that h is a homomorphism from the space of IGL functions and domain to the corresponding FGL space. Here we may rely on the dag forms of the corresponding IGL and FGL programs. The technique is essentially that explained by [18]. [11] presents a similar theorem, stated in terms of a "semantic memory" instead of FGL arc values.

Since it is generally meaningless to speak of an evaluator producing a full FGL object, we phrase our definition of evaluator correctness in terms of IGL objects, as follows:

IGL Partial Correctness Theorem: If q is a state and x an arc marked demanded in q, and $\text{Ival}(x) \neq ?$, then the IGL evaluator can reach a state q' such that x is marked with its corresponding IGL value.

To justify this theorem, we identify node x as the node having x as its output arc. Consider the corresponding dag structure of the IGL graph with root node x, assuming now that $\text{Ival}(x) \neq ?$. Then either node x is a constant function having value Ival(x), or x produces Ival(x) based on the values of its arguments. In the first case, one transition rule gives us the desired result. In the second case, the inductive assumption is that the arguments evaluate appropriately so that evaluating the function in node x gives the desired result. Thus, the inductive conclusion tells us that these arguments can be produced by application of the transition rules. Therefore application of one or more transition rules for the root node will produce Ival(x).

The above use of induction is technically justified from the continuity of IGL operators. Informally, this says that a finite value (e.g. any IGL value) producible from an arbitrary composition of operators is also producible from a finite truncation of that composition. For a further discussion of such uses of continuity, see [19], [20], or [16].

Given this partial correctness, we have the corollary that any finite piece of an IGL value can be produced by an appropriate set of demands. Simply affix to the arc in question a supplementary function graph of selectors which evaluate to that piece formally, then apply the above criterion to the output of the supplementary graph.

By assuming that the underlying IGL evaluator has the finite-delay property, the "can" above effectively becomes "will". This property is provided in the definition of ML. This approach is necessary since there is no mechanism for insuring the finite-delay property within IGL itself.

In the next section, we appeal to ML to provide the necessary infrastructure for total correctness of the IGL evaluator.

ML TO IGL MAPPING

As stated earlier, ML and IGL have the same data objects. As the corresponding ML-->IGL mapping is rather trivial, involving only replacement of linkage operators by identities, it will not be elaborated upon here. Furthermore, both IGL and ML are evaluated by changing the operations of their nodes into values. In ML however, we provide an implementation of demand/value propagation symbolized by markings in IGL.

In IGL, the presence of a demand for a node's value is indicated by marking the node with an asterisk. It would be infeasible, in ML, to search the memory for all demanded nodes each time a new value is computed. Instead, ML employs a task list structure which contains pointers to all nodes on the wavefront of demand/value propagation (see Figure 6). The wavefront may be thought of as initially propagating in the direction opposite to the argument arrows and being reflected in the opposite direction when computed values are encountered.

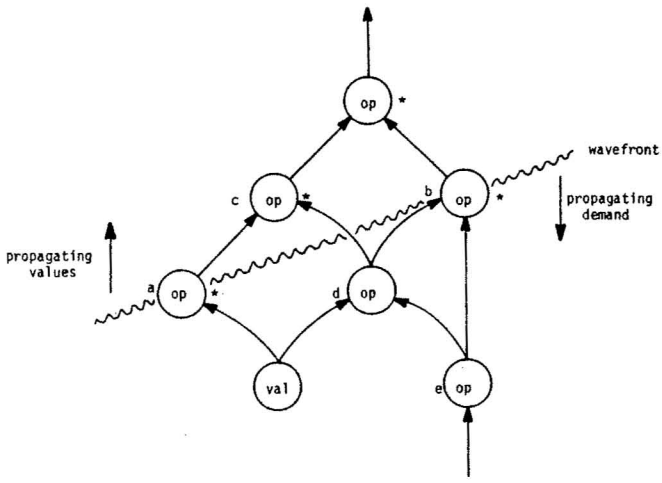

Figure 6: Wavefront of demand/value propagation. Nodes a and b are currently on the task list. a will be evaluated and notify c; b will propagate its demand to d and e.

For demanded nodes not on this wavefront, the fact that the node has been demanded is recorded by the presence of a notifier or a forward pointer (see next section) in some other demanded node. Thus, consider the following definition of a set of nodes S:

1. Nodes on the task list which do not yet have a value are in S.[a]
2. If x is a node in S, and x contains a notifier or forward pointer to node y, then y is in S.
3. All nodes in S are there because of one of the above reasons.

Wavefront Lemma: S consists of exactly the demanded nodes.

The proof of the above lemma is by transition induction (cf. [12]) on the ML transition rules. All initially demanded ML nodes are externally placed on the task list. A case analysis of the ML transition rules reveals that any newly demanded node is put in S. Similarly, any node which is replaced with its value cannot remain in S, but nodes requiring that value are put in S.

We repeat that the finite-delay property for IGL means that every demanded node in a given state, if entitled to eventually receive a value (because the IGL output value of that node is not ?), will receive a value. As is well known, FIFO processing of nodes in a directed graph gives rise to breadth-first visitation of the nodes, i.e. the wavefront effect. By processing the task last in FIFO order, it is clear that any node in need of attention eventually receives that attention. In particular, every node gets attention when it is first demanded, and when it is able to compute its value.

In the proposed AMPS architecture, the task list is not monolithic, but instead is distributed among many processing elements. However, each of the segments is processed in FIFO order, so the same wavefront effect is obtained.

PRAGMATIC ASPECTS

Although not required for correctness as stated, parsimonious evaluation is also achieved. That is, each node is evaluated at most once, since the presence of a notifier inhibits potential secondary demand propagation. This idea, applied to the cons operator, was called "suicidal suspension" in [10]. It has also been used in operating systems (e.g. the dynamic linking mechanism of Multics) for some time. Our evaluator includes this technique for all operators.

ML includes additional operators apart from IGL, namely the special operators used to control data flow across block boundaries. Specifically, whenever a selector in one block refers to a tuple in another, the selector is replaced with the special fetch operator which matches a forward operator in the tuple component. The fetch operator contains the global address of the forward. A demand of the fetch (which occurs

[a] Because of some redundancy in the evaluator, a node could be on the task list and have a value. For example, it could be notified by two different nodes, and become evaluated before the second notification "takes effect".

automatically when the selector is demanded) is then propagated to the **forward**, which propagates the demand to another operator local to its block. At the same time, a forward pointer back to the **fetch** is set to point to the **forward**, so that when the demand is satisfied, the forward will know where to send the result. A **fetch/forward** pair is also used to pass the result of the block to its destination.

A possible alternative to forward chaining is to use "busy waiting". That is, the second and subsequent fetches for the same value are simply re-cycled back to the task list to be re-tried again and again. This solution is viewed as unacceptable, as the wait can be arbitrarily long.

As described thus far, the ML **fetch/forward** pairs resemble identity functions which carry out the linkage needed to implement an arc crossing blocks in IGL. However, a complication arises when there is more than one demand on the same component of a tuple. This complication was not mentioned in [10] where it does not occur because evaluation is sequential, but neither was it mentioned in [7]. The property asserted there of the existence of at most one reference to any "suspension" seems infeasible for a parallel evaluator, as we now discuss.

Since the number of demands may, in principle, be arbitrary, there is no fixed word size which can accommodate sufficiently many forward pointers. Hence a scheme called forward chaining is used. This scheme maintains the invariant (provable by transition induction) that at most one forward pointer is ever stored in a given **forward** node. This is accomplished by having each additional fetch to the same **forward** operator assume the responsibility for forwarding to the location to which the **forward** pointer pointed, while the forward operator then points to the most recent fetch only. The handling of **fetch** and **forward** in ML is demonstrated in Figure 7.

Although there is no limit on the number of (local) notifiers a node may entail, the number actually needed in each case can be detected at compile time. Hence it is possible for the compiler to cascade extra identity operators in such a way that the number of notifiers for each node does not exceed the maximum pragmatically allowed.

CONCLUSIONS

We have described some considerations which arise in the evaluation of an applicative language in a manner capable of exploiting a multiplicity of physical processing elements. The present exposition focuses on the analysis of a hardware evaluator for the AMPS system. In addition to the graphically-represented extrinsic language and machine language, an intermediate graphical language has been introduced, to separate questions of value flow from more pragmatic issues of communication and demand flow.

The important aspects of this work thus concern the distributed evaluator itself, the analysis techniques, the graphical models, the formalization of demand-driven computation and accompanying correctness criterion, and further technical exposition of machine evaluation of unbounded data objects.

We view this analysis as a step toward a proof for a fuller system in which a reference-counting storage manager is implemented (cf. [21]), as well as other language and pragmatic issues, such as shared resource management and load control [17].

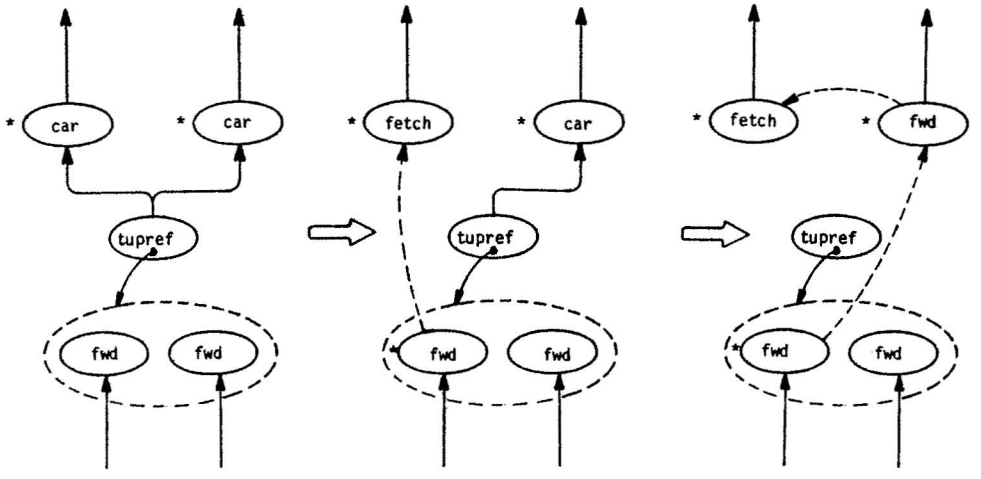

Figure 7: Example of forward chaining in ML.
Hyphenated arcs denoted global addresses.

REFERENCES

[1] G. Brown. A new concept in programming. in M. Greenberger (ed.), Management and the computer of the future. Wiley (1962).

[2] S. Patil. An abstract parallel-processing system. M.S. Thesis, MIT Dept. of Electrical Engineering (June 196).

[3] L.L. Constantine. Control of sequence and parallelism modular programs. AFIPS Proc., 409-414 (Spring 1968).

[4] L.G. Tesler and H.J. Enea. A language design for concurrent processes. AFIPS Proc., 403-408 (Spring 1968).

[5] D.A. Adams. A model for parallel computations. in Parallel processor systems, technologies, and applications. Spartan Books, 311-333 (190).

[6] W.H. Burge. Recursive programming techniques. Addison-Wesley (195).

[7] D.P. Friedman and D.S. Wise. The impact of applicative programming on multiprocessing. IEEE Trans. on Computers, C-2, 4, 289-296 (April 198).

[8] R.M. Keller, G. Lindstrom, and S. Patil. A loosely-coupled applicative multi-processing system. AFIPS Proc. (June 199).

[9] R.M. Keller, B. Jayaraman, G. Lindstrom, J.B. Marti, A.K. Nori, and D. Rose. FGL Programmers' Guide. Unpublished manuscript, University of Utah (March 1980).

[10] D.P. Friedman and D.S. Wise. CONS should not evaluate its arguments. in Michaelson and Milner (eds.), Automata, Languages, and Programming, 25-284, Edinburgh University Press (196).

[11] P. Henderson and J.H. Morris, Jr. A lazy evaluator. Proc. Third ACM Conference on Principles of Programming Languages, 95-103 (196).

[12] R.M. Keller. Formal verification of parallel programs. CACM, 19, , 31-384 (July 196).

[13] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, I. CACM, 5, 2-3 (190).

[14] A. Church. The calculi of lambda-conversion. Princeton University Press (1941).

[15] R.M. Keller. Divide and CONCer: Data structuring for applicative multiprocessing systems. to appear in Proc. 1980 Lisp Conference.

[16] R.M. Keller. Semantics and applications of function graphs. Manuscript (March 1980).

[17] B. Jayaraman and R.M. Keller. Resource control in a demand-driven data-flow model. Proc. International Conference on Parallel Processing (Aug. 1980).

[18] C.A.R. Hoare. Proof of correctness of data representations. Acta Informatica, 1, 21-281 (192).

[19] G. Kahn. The semantics of a simple language for parallel programming. Proc. IFIP '74, 41-45 (194).

[20] J. Stoy. The Scott-Strachey approach to the mathematical semantics of programming languages. MIT Press (19).

[21] A.K. Nori. A storage reclamation scheme for AMPS. M.S. Thesis, Dept. of Computer Science, University of Utah (Dec. 199).

[22] M. O'Donnell. Computing in systems described by equations. Lecture Notes in Computer Science, 58 (19).

This material is based upon work supported by the National Science Foundation under grants MCS 77-09369 A01 and MCS 78-03832.