Proc HICSS 1989

Lispflows: Specification of a Model for Functional Programs

David L. Coleman coleman@atanasoff.cs.iastate.edu
Albert L. Baker[1] baker@atanasoff.cs.iastate.edu
James M. Bieman bieman@atanasoff.cs.iastate.edu

Department of Computer Science, Iowa State University, Ames, Iowa 50011, (515) 294-4377

Abstract

Software development tools and measures are often based on particular models of programs. For example, flowgraphs serve as a model of imperative programs. There is no analogous, commonly accepted model of functional programs.

The benefits of models of programs are clear. Flowgraphs serve as the basis for numerous tools, e.g., testing environments that monitor structural coverage criteria. Flowgraphs are also the basis for numerous measures of control structure, data dependency, and program size.

The increased use of functional languages motivates development of a model for the functional paradigm. Lispflows model functional programs in the same sense that the flowgraphs model imperative programs.

Lispflows are based on dataflow graphs. We define lispflow as an abstract data type (specifically as an abstract object), using abstract model specifications. A mapping from a full dialect of Pure LISP to lispflows is described. Additionally, we define several properties on lispflows that are (loosely) analogous to existing properties for flowgraphs.

Lispflows lend insight into the recursive structure and functional composition of functional programs. Our initial comparisons suggest that lispflows may be a generalization of the models used for imperative programs and may thus serve as a basis for defining tools and measures applicable to a broad category of programming paradigms.

1 Introduction

In this paper we define and provide example uses of a dataflow model of functional language programs. This model, a "lispflow", represents functional programs in much the same way flowgraphs model imperative programs. The lispflow model provides insight into the fundamental structure of functional language programs. Thus it may be used to define software tools and software measures for functional language programs, analogous to similar flowgraph-based tools for imperative language programs. We define several structures based on the lispflow model of functional programs. The defined structures demonstrate the utility of the dataflow model.

We consider only pure functional programs. A pure functional program is a composition of function calls without side effects. The lispflow model is defined for a side effect free LISP subset (without sequencing of expressions or iterative constructs[2]). In the conclusion we address the generalization of the lispflow model to production dialects of LISP.

Since the use of functional languages for commercial and large scale software projects is increasing, and since the functional paradigm has expressive advantages for software system specification and design [1], a general, well-defined model for the functional paradigm has timely significance. The professional community will only be able to make effective use of these functional specification and design techniques when CASE tools to support the functional paradigm are readily available. Flowgraphs for the imperative paradigm help support design and analysis of imperative software. Lispflows can serve an analogous role for the functional paradigm.

Flowgraphs for imperative language programs also serve a variety of other roles, including code optimization strategies, structural testing criteria, and software measures. The lispflow model of functional language programs can be used to define analogous tools and software measures. These tools and measures can also be used to manage and analyze software development based on the functional paradigm.

The flowgraph model of imperative language programs and the lispflow model of functional language programs are necessarily based on the static (compile-time) structure of programs. The treatment of code as data, i.e., **quote**, **eval**, **apply**, and **funcall** in LISP, is important for A.I. applications, but is problematic for static representations. The lispflow model defined in Section 4 addresses the code as data feature of functional languages. Such features in imperative languages are not handled by the widely used flowgraph models.

Several fundamental differences between the functional and imperative paradigms prevent flowgraphs from being effective models of functional language programs:

1. Functional programs do not consist of sequentially executed commands, but rather use patterns of nested function applications. The result of a nested function is used immediately by the function it is nested within. Thus the flow in a functional language program is of values in expressions, not of control between commands.
2. Pure functional languages lack the assignment operator and parameter passage by reference.
3. Imperative programs use conditional transfer of control; the value of a predicate in the current environ-

[1]Address Correspondence to Dr. Albert L. Baker, Department of Computer Science, Iowa State University, Ames, IA, 50011.

[2]For a detailed development of the specific subset of LISP, see [5].

ment determines which statement is to be executed next. On the other hand, a functional language program uses conditional expressions to determine which function value to return. As we shall see in Section 3, composition of functions is represented as incoming flows. Thus a lispflow conditional node has several incoming edges and produces a single answer edge. A flowgraph conditional node has one or more incoming control edges and passes control on to one of several outgoing edges.

4. Pure functional languages lack an explicit functional iterative construct. In a functional language program we are restricted to the use of recursion for any repetitive process. Since the lispflow model represents functional language programs at the unit level, there can be no loops in the dataflow graph for a particular defined function.

A brief introduction to dataflow programs as originally described by Dennis [6] is contained in Section 2. In Section 3 we give an informal example of a lispflow. In Section 4 we give a careful definition of lispflow. In Section 5 we describe the algorithm for generating the lispflow representation of a pure LISP program. Several structural properties of lispflows are defined in Section 6. The analogy with features of flowgraphs is described, where appropriate. As part of our conclusions in Section 7 we speculate on the usefulness of the dataflow approach to modeling more general dialects of LISP and comment on other applications.

2 Comments on Dataflow

The basic concepts of the dataflow paradigm are sufficiently important for the lispflow model that a few brief comments may be helpful. There are numerous variations on the dataflow paradigm and a similarly large number of applications. One application is in the software specification technique commonly referred to as "structured analysis"[11]. An integral part of such a specification is a hierarchy of diagrams which depict the flow of information in the corresponding executable system. While this application does not utilize the notion of control used in dataflow programs, it does serve to emphasize a major feature of dataflow — the movement of information (or data).

The application of the dataflow paradigm in a programming language is due to Dennis[6]. Dennis depicts a dataflow program as a directed graph where the edges represent pipelines along which information flows and the nodes represent transformations of incoming flows into outgoing flows. When sufficient incoming flows have reached a node and there is currently no outgoing flow, the node is said to fire. When a node fires, it consumes the incoming flows and produces an outgoing flow. Firing rules for each type of node describe when sufficient incoming flows are present.

Dennis uses a form of heap to allow for structured value flows. The actual tokens flowing on the edges in Dennis' language are pointers to nodes in the heap. The heap is a finite, acyclic, directed graph having one or more root nodes. This heap representation of structured data can be used to model the available LISP data types [5].

Dennis distinguishes two types of information flows. One type is a flow that carries data. Data flows carry tokens that represent either an elementary or structured value. The other type of flow carries control information. Control information is of type boolean. Edges in Dennis' dataflow language that represent control information flows are identified by open circles and open arrows. Edges that represent data flows are identified by filled circles and filled arrows. We retain this open/closed circle convention in the lispflow model.

The nodes in Dennis' dataflow language are either called links or actors. A link node simply produces multiple copies of a single incoming flow. Dennis defines a number of different actor nodes, e.g., an operator actor which produces a single data flow, a decider actor which produces control information, and various constructor and selector actors for structured data flows. For a complete list of the Dennis dataflow operators needed to represent LISP programs, see [5]. This particular set of dataflow nodes is theoretically adequate for modeling Pure LISP programs. However, the resulting dataflow representations are cumbersome, and hence not very expressive from the standpoint of the human reader. The lispflow model described in Sections 3 and 4 uses the basic concepts of dataflow in a more expressive model of functional programs.

3 An Informal Lispflow Example

A simple example illustrates the basic concepts of the lispflow model of functional programs and serves as a preface to the more formal definition appearing in Section 4. The LISP function Member appearing in Figure 1 returns t (true) if the atom A is a member of the list of atoms Lat, and nil otherwise.

```
(defun Member (Lat A)
       (cond ((null Lat) nil)
             ((eql (car Lat) A) t)
             (t (Member (cdr Lat) A))
)      )
```

Figure 1: Simple LISP Function Member

A lispnow models an entire LISP program which is composed of functions like Member. Each function in a LISP program is modeled by a single *FuncGraph* element of a lispflow. The FuncGraph for the simple LISP function Member appears as Figure 2.

FuncGraphs are analogous to Dennis' dataflow programs in that the edges in a FuncGraph represent paths on which data values flow. The nodes in the example FuncGraph labeled 1 and 2 represent sources of the parameter values Lat and A, respectively. "NORMAL" refers to the type of LISP parameter and is not significant for this example. The node labeled Member at the bottom of the FuncGraph represents the return value of the function Member. The remainder of the FuncGraph models the body of the LISP function Member.

Since the body of the function is a single LISP conditional operator cond, first consider the special *control node* 12 in Figure 2. The sequence of cells 1, 2, and 3 in the cond control node represent the three clauses of the cond operator appearing in Member. The edge into each cell that terminates in an open circle represents the predicate of the corresponding clause. We refer to these as *p-use* edges. For example the predicate of the first clause of the cond operator in Member is (null Lat). This predicate is modeled by the path from the parameter node 1 through the *value node* 3 terminating in the p-use edge into cell 1 of node 12.

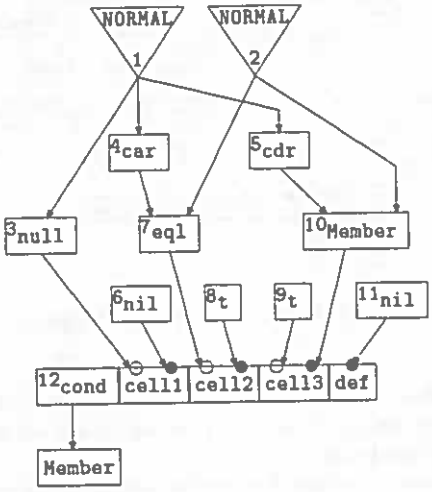

Figure 2: FuncGraph for Function Member

The edge into each cell of a control node that terminates in a closed circle represents the return value associated with the corresponding clause. We refer to these as *d-use* edges. The return value of the first clause of the cond is just nil. This is represented by the d-use edge from the value node 6 to cell 1 in node 12. Note that the value that flows out of a cond control node will be the value on the d-use edge into cell i where the value on the p-use edge into cell i is non-nil, and, for every j, $1 \leq j < i$, the value on the p-use edge into cell j is nil. This simply models the semantics of LISP control operators like cond.

In Section 6 we use the distinction between p-use and d-use edges to develop certain structural properties of FuncGraphs based on paths. A path that terminates in a p-use edge is called a *predicate path*. Paths that do not terminate in p-use edges must terminate at the result node. We call these *value paths*. These notions are developed more fully in Section 6.

4 Lispflows

In this Section we provide a formal definition of the lispflow model of syntactically correct Pure LISP programs. Lispflow is defined as an abstract data type domain using abstract model specifications[2]. The domain of an abstract data type is a collection of objects defined by a source set and an invariant over the source set. The source set is presented as a series of type definitions that describe the structural nature of a lispflow. (The abstract model specification language we use has primitive structured types finite set, finite sequence, and n-tuple[2] analogous to other specification languages like VDM[9].) Any object with the structure of the source set that satisfies the invariant is a valid instance of a lispflow.

A lispflow is a collection of dataflow-like graphs. Nodes of these graphs can represent Pure LISP operators, function applications, or parameter references. Each edge in a graph represents the flow of an argument value to the node representing the operator to which the argument is applied. Nodes representing function application may also require a functional form argument.

The lispflow model is based on the following definition of a Pure LISP program: a *Pure LISP program* is a set of Pure LISP function definitions which may not refer to other function definitions not in the set. Thus a Pure LISP program is closed under the "calls" relation as far as we can statically determine.

Type Definition 1 formally defines a lispflow.

Type Definition 1

LISPFLOW = set of FuncGraph;

Each FuncGraph in a lispflow models one Pure LISP function definition. A FuncGraph is an acyclic directed graph that represents a dataflow mapping from an ordered sequence of parameters to a result. The nodes of a FuncGraph represent Pure LISP parameters, operators, constants, and application of functional forms.

Each parameter node in a FuncGraph behaves like one of Dennis' dataflow link nodes. An edge from a parameter node goes to every node that represents an operator or a function application that references that formal parameter[3]. Edges representing Pure LISP composition go from these operator and function application nodes to other operator and function application nodes until a final edge connects to the FuncGraph's result node. By distinguishing these operator and function application nodes as specific node types, we can characterize the movement and use of data through a Pure LISP function.

We classify nodes in a FuncGraph as either parameter nodes, application nodes, value nodes, or control nodes. We also declare a special node called a result node. Type Definition 2 defines a FuncGraph as a 6-tuple. The *FuncId* of a FuncGraph identifies the FuncGraph and the result node. The result node is the only node with out-degree 0 and in-degree 1.

Type Definition 2

FuncGraph = 6-tuple(*FuncId*:IdType,
Parameters:
sequence of ParamNode,
Applys:set of ApplyNode,
Values:set of ValueNode,
Controls:set of ControlNode,
Edges:set of EdgeType);

[3] We assume every formal parameter is referenced at least once in the body of the function definition.

We frequently refer to the example given in Figures 3 and 4. The LISP function in Figure 3 evaluates a list representation of a prefix arithmetic expression. The corresponding FuncGraph appears as Figure 4. Note that the FuncGraph for PrefixEval contains a single parameter node and that the result node is labeled with the function name.

For clarity in the following discussion, we refer to an edge as an *argument edge* if that edge is in-coming to an application, value, or control node. Likewise, we refer to an edge as a *result edge* if that edge is out-going from an application, value, or control node. An edge is then an argument or result with respect to a particular node.

The parameter and application nodes represent the interface between defined functions in a Pure LISP program. Parameter nodes are ordered consistent with the formal parameters they represent. Argument edges to an application node are numbered to represent the order of the actual parameters. The ordering of parameter nodes and the argument edges of application nodes allows us to represent the binding

```
(defun PrefixEval (Exp)
        (cond ((null Exp) nil)
              ((null (cdr Exp)) Exp)
              ((numberp (car Exp))
               (cons (car Exp) (PrefixEval (cdr Exp))))
              ((and (numberp (car (cdr Exp)))
                    (numberp (car (cdr (cdr Exp)))))
               (cons (eval (list (car Exp)
                                 (car (cdr Exp))
                                 (car (cdr (cdr Exp)))))
                     (PrefixEval (cdr (cdr (cdr Exp))))))
              (t (PrefixEval (cons (car Exp)
                                   (PrefixEval (cdr Exp)))))
)       )
```

Figure 3: Example LISP Function PrefixEval

of actual to formal parameters. This serves to model data dependencies across a functional interface.

Operator nodes are separated into control and value nodes. A control operator differs from a value operator in two basic ways. First, a control operator does not compute the value it returns. The control operator simply determines which of a number of data values on argument edges will be passed on as the result of the control node. Second, not all arguments to a control operator are used to form the result of the operator. This may seem redundant since we just said the result of a control operator originates from outside the control operator, but the point is that not all arguments to a control operator are possible results. This second difference relates to the notion of p-uses and d-uses[4] defined by Rapps and Weyuker[12]. We say that an edge is a p-use edge if it is used to select a value to be returned by the control node. Alternatively, a d-use edge represents a value that may be selected. For example, the LISP and operator is represented by a control node since it selects its results from its arguments.

[4] p-use and d-use stand for predicate use and definition use, respectively.

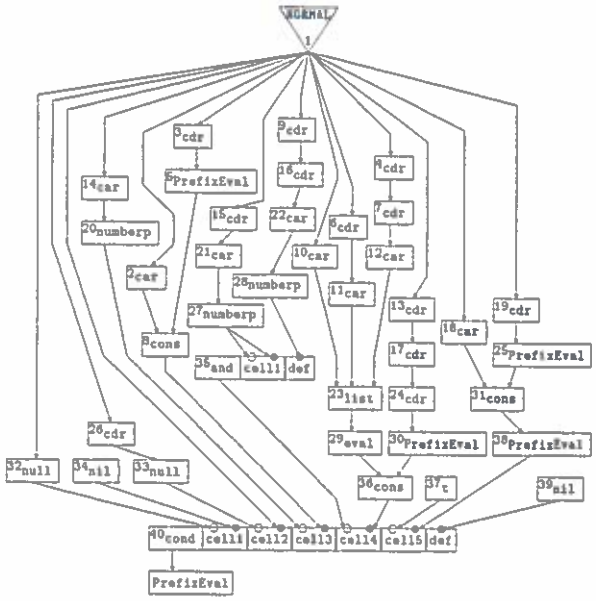

Figure 4: FuncGraph for Function PrefixEval

See node 35 in Figure 4. Also note that the single node 27 serves as the source of both the p-use and d-use edges into cell 1 of this and node.

Each formal parameter is represented by a parameter node labeled by a unique identifier. The out-degree of a parameter node is equal to the number of references to the formal parameter in the function definition. Type Definition 3 formally defines a parameter node.

Type Definition 3

ParamNode = 2-tuple(*NodeId*:IdType,
Type:ParamType);

ParamNode identifies each parameter node with a *NodeId* and *Type*. In our PrefixEval example, the name of the formal parameter could be used as the *NodeId* value. A parameter's *Type* is defined to be one of the available LISP parameter types NORMAL, OPTIONAL, or REST. Type Definition 4 formally defines the available *Types* of a parameter node.

Type Definition 4

ParamType = (NORMAL,OPTIONAL,REST);

Each occurrence of a function application is represented by an application node. Type Definition 5 formally defines the contents of an application node.

Type Definition 5

ApplyNode = 3-tuple(*NodeId*:IdType,
FuncId:FuncType,
NumParams:NonNegativeInteger);

The *NodeId* uniquely identifies an application node. The *FuncId* is either the identity of a known function represented

by a FuncGraph or is an unknown function. Type Definitions 6, 7, and 8 define the representation of the *FuncId*.

Type Definition 6

FuncType = Known |Unknown;

Type Definition 7

Known = IdType;

Type Definition 8

Unknown = 0-tuple( );

The identity of an unknown function is run-time dependent and cannot be statically determined. An application node of an unknown function requires one more argument edge than an application node of a known function. This extra edge supplies the functional form of the unknown function. *NumParams* contains the number of actual arguments to an application of a function. Since the number of arguments can vary for applications of the same function, this number may vary between application nodes with equal known *FuncIds*. If *NumParams* is 0 then this represents an application of a constant function or a function accepting a variable number of arguments.

Value nodes represent Pure LISP value operators and Pure LISP constants. Value nodes are represented similarly to application nodes. Type Definition 9 formally defines a value node.

Type Definition 9

ValueNode = 3-tuple(*NodeId*:IdType,
OpId:IdType,
NumArgs:NonNegativeInteger);

Each value node is uniquely identified in the FuncGraph by its *NodeId*. Value nodes representing occurrences of the same Pure LISP value operator will share a common *OpId*. *NumArgs* contains the number of arguments to this occurrence of the Pure LISP operator. In the FuncGraph in Figure 4 node 23 is a value operator with OpId **list** and NumArgs equal to 3. If *NumArgs* is 0 then the node represents a Pure LISP constant, a **quote**, or **lambda** operator.

A control node represents any of the possible control operators provided for in the Pure LISP dialect described previously[5]. The generalization of the different control operators into one node type gives us a powerful node description.

Each LISP control operator can be thought of as a sequence of boolean/value pairs. The semantics of the LISP operator is determined by how this sequence is treated. In the lispflow representation of these conditional operators we consider each boolean/value pair as a p-use/d-use cell, and we simply number the cells. Identifying edges into the cells is handled in the destination information of each edge and is described subsequently. Type Definition 10 formally defines a control node.

[5]For a detailed development of the adequacy of the control operator for representing all the control operators in the specified dialect of Pure LISP, see [5]

Type Definition 10

ControlNode = 4-tuple(*NodeId*:IdType,
OpId:IdType,
NumCells:PositiveInteger,
Key:Boolean);

Again, the *NodeId* uniquely identifies each control node from all the other nodes in the FuncGraph. The *OpId* identifies occurrences of the same control operator. *NumCells* is the number of p-use/d-use cells. The key edge is required to represent the *key* argument in the **case** operator. If the control node represents a control operator requiring a key p-use edge, then *Key* is true, otherwise *Key* is false. The key p-use edge will result in one more p-use argument edge than the number of p-use/d-use cells. Default values are handled by having one more d-use argument edge than there are p-use/d-use cells. The default value of a **cond** is **nil** and we represent the default edge source as a constant value node with *OpId* **nil**. In the FuncGraph for PrefixEval, node 40 is a **cond** node and node 39 is the source of the default d-use edge.

Each node type describes the number and type of arguments required for an instance of a node. Each edge has a specified source and destination node and represents a connection from one node to another in the FuncGraph. To identify the source of an edge we need only the node identifier. The destination of an edge could be a control node, a value node, an application node, or the result node. To identify the result node as the destination requires only the FuncGraph identifier. Edges destined for an application node or value node require an argument number in addition to a node identifier. Finally, edges destined for a control node must be identified as either a key or default edge or as a p-use or d-use with a corresponding cell number. Type Definition 11 formally defines an edge.

Type Definition 11

EdgeType = 2-tuple(*From*:IdType,
To:ToType);

The *To* portion of an edge is dependent on the node type for which the edge is an argument. Type Definition 12 alternatively defines the destination of an edge for the various node types.

Type Definition 12

ToType = ApplyArg |ValueArg |ControlArg |Result;

Each alternative ToType describes the information necessary to specify any particular edge destination. Type Definition 13 defines an application node destination.

Type Definition 13

ApplyArg = 2-tuple(*NodeId*:IdType,
ParamNum:NonNegativeInteger);

ParamNum is the argument number for this edge. This number must be between 0 and the *NumParam* (inclusive) of the destination application node. If the *ParamNum* for

the edge is 0, then the application node represents an application of an unknown function, and this edge represents the functional form or operator to be applied.

Type Definition 14 defines a value node destination.

Type Definition 14

ValueArg = 2-tuple(*NodeId*:IdType,
ArgNum:PositiveInteger);

The *ArgNum* is the argument number for this edges destination. *ArgNum* must be between 1 and the *Num.Args* (inclusive) of the value node.

Type Definition 15 defines the destination of an edge as a control node.

Type Definition 15

ControlArg = 2-tuple(*NodeId*:IdType,
ArgNum:PositiveInteger,
ArgUse:UseType);

If the edge is a key p-use edge then *ArgUse* equals *PUse* and *ArgNum* equals 0. If the edge is the p-use edge of a cell then *ArgUse* again equals *PUse* and *ArgNum* indicates the cell number. If the edge is the default argument edge then *ArgUse* equals *DUse* and *ArgNum* equals 0. If the edge is the d-use edge of a cell then *ArgUse* also equals *DUse* and *ArgNum* indicates the cell number. Type Definition 16 formally defines UseType.

Type Definition 16

UseType = (PUse, DUse);

If the value on an edge represents the result of the FuncGraph then the *To* of the edge is the identifier, i.e. *FuncId*, of the FuncGraph. Type Definition 17 defines an edge destination as the result node.

Type Definition 17

Result = IdType;

IDType can be any type for which equality is defined and we leave the specification of this type to the implementer:

Type Definition 18

IdType = generic;

In this paper we instantiate *IdType* as type string. We use the common practice of identifying nodes by a number (a string of digits), except for result nodes which we identify using the name of the function.

We assume the usual specifications of PositiveInteger and NonNegativeInteger.

The source set described by the type definitions is a set of objects. Not all of these objects are representations of Pure LISP programs. In order to complete the domain specification we would need to enumerate the various properties that all valid lispflows must satisfy. These properties include the uniqueness of NodeId's in a given FuncGraph and that the argument number of an edge into a value node must not exceed the number of arguments for that node (and that each argument edge has a unique argument number). A complete and formal enumeration of these properties are in [5].

5 Mapping Pure LISP to Lispflow

Lispflow models each LISP function definition as a FuncGraph. The mapping from a Pure LISP program to the lispflow model depends primarily on the mapping from a Pure LISP function to a FuncGraph. We assume function definitions are syntactically correct and that we can identify any particular literal atom as either a Pure LISP operator (either control or value) or as the name of a defined function.

The first phase of the mapping sets up the environment for modeling the body of a Pure LISP function. For each function definition of the form (**defun** <name> (X1 X2 ... Xn) <function-body>) we define a FuncGraph F with $FuncId(F)$ equal to <name>. $Parameters(F)$ is defined as the sequence of parameter nodes representing X1 X2 ... Xn with the appropriate parameter node types, i.e., NORMAL, OPTIONAL, or REST. We define a result edge E which will connect the representation of the s-expression <function-body> to the result node, i.e. $To(E) = FuncId(F)$.

The mapping of <function-body> to the FuncGraph representation is based on the functional composition and the recursive nature of the LISP evaluation rule. The mapping's second phase maps an s-expression S (initially the <function-body>) to a FuncGraph representation with result edge E (initially the result edge of the FuncGraph from phase one).

If the s-expression S is atomic then it must either be a constant or parameter reference. A constant is represented by defining a value node N with no argument edges and setting $From(E) = NodeId(N)$. A parameter reference is represented by setting $From(E)$ equal to the $NodeId$ of the parameter node representing the parameter referenced.

If the s-expression S is a non-empty list then the FuncGraph representation of S is based on one of the following mutually exclusive cases[6]:

Case 1: If the first element of S is a list then this is an instance of an application of an unknown function. We define an application node N with $FuncId(N)$ unknown and set $From(E) = NodeId(N)$. For each i^{th} element of S an argument edge E_{i-1} is defined such that $NodeId(To(E_{i-1})) = NodeId(N)$ and $ParamNum(To(E_{i-1})) = i-1$. The phase two mappings of s-expressions is recursively applied to the i^{th} element of S with result edge E_{i-1}.

Case 2: If the first element is atomic then it must be either a value operator, known function, or a control operator. Value operator and known function s-expressions are handled very much like Case 1 above, except there will be no E_0 argument edge since the node defined is either a value node or an application node with a known $FuncId$. For a control operator s-expression, two argument edges, one p-use and one d-use, are defined for each clause. If a clause is atomic (as in the and operator) or has length 1, then the representation of that

[6] A non-list dotted pair is treated as a constant since dotted pairs represent a constant structured value and cannot be evaluated.

clause serves as the source for both argument edges. Otherwise, the representations of the car and cadr of the clause serve as the source for the p-use and d-use argument edges, respectively. Based on the particular control operator, a representation for a key argument may be required. In every control operator a default argument, either implicit or explicit, is provided and must be represented with a default edge with appropriate source.

The described algorithm translates a Pure LISP function definition into a FuncGraph. Certain aspects of a FuncGraph's construction may not be intuitively acceptable from a programmer's point of view. For instance, frequently the last p-use argument of a cond or case operator is the symbol t. Then, the default value nil of the cond or case operator could never be realized as an actual returned value. In that case, it may be more appropriate to ignore the t p-use argument and use its corresponding d-use argument as the default edge for the control node. Similarly, any constant valued s-expression, e.g., (ex. (+ 3 4)), can be evaluated and the representation of its value is used in place of the representation of the constant valued s-expression. This modification is carried out across function interfaces to produce a "most simplified" yet semantically equivalent lispflow.

Finally, multiple occurrences of the same s-expression can be combined into one representation and multiple result edges are used to supply each destination node with the argument edge required. For example, consider nodes 3, 4, 6, and 9 in the PrefixEval FuncGraph in Figure 4. Each of these nodes has OpId **cdr** with exactly the same argument. Therefore these nodes can be coalesced into a single node, namely the node 2 in the revised FuncGraph in Figure 5. Note that node 2 in this revised FuncGraph must have multiple result edges, one for each result edge for the nodes 3, 4, 6, and 9 in the original FuncGraph. The FuncGraph of Figure 5 reflects the enhancement transformations possible given the original FuncGraph of Figure 4.

6 Definitions of LISPFLOW Structures

We now use the FuncGraph model to define several structures that underlie Pure LISP functions and suggest how these might be used to define tools and measures analogous to tools and measures based on flowgraphs. The perspective of edges as value carriers and nodes as value producers is consistent with a view of the FuncGraph as an executable representation of the function definition.

We can identify and define several notions of paths in a FuncGraph. A path represents a data dependency relation on a sequence of edges. These dependencies may be either value or control dependencies. We can then define collections of paths that represent executions of parts of the FuncGraph. These definitions culminate in a definition of an execution trace of a Pure LISP function analogous to the definition of an execution path in an imperative procedure.

In a FuncGraph, a path represents a flow of values from one node to another. This flow represents a data dependency where the next result produced is dependent on the results of the previous nodes in the path. Node types determine

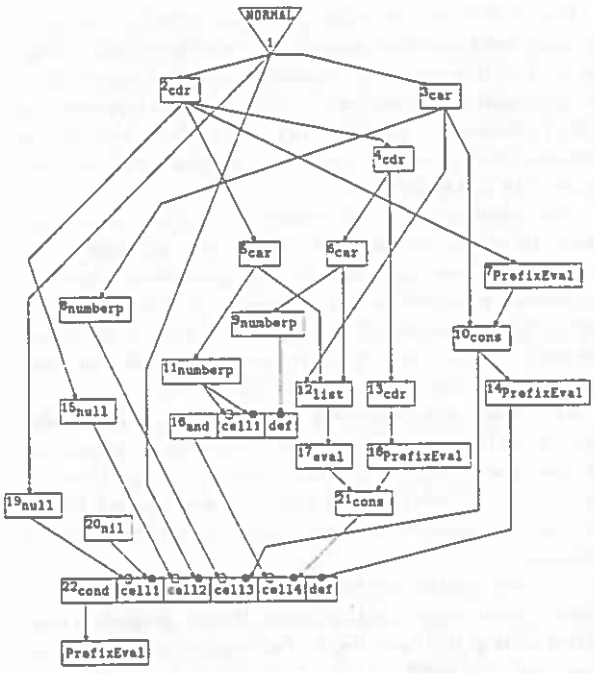

Figure 5: Modified FuncGraph of Function PrefixEval

whether a dependency is a control dependency or a value dependency. Since data values always originate at parameter or constant value nodes, we define a path as starting from one of these nodes. Other than the starting point restriction, a path in a FuncGraph can be defined in a standard manner as a sequence of edges.

Control nodes in a FuncGraph are obviously the source of control structure. A p-use argument value predicates the flow of d-use argument values through a control node. The choice of a d-use argument edge E_d is control-dependent on all p-use argument edges E_p with $NodeId(To(E_p)) = NodeId(To(E_d))$ and $ArgNum(To(E_p)) \leq ArgNum(To(E_d))$. We define a *predicate path* as any path in a FuncGraph that ends with a p-use argument edge of a control node and does not contain any other p-use edges. Thus a predicate path does not properly contain another predicate path. However, two predicate paths may share common edges including the final p-use edge. We use the expression $PredPaths(F)$ to denote the set of all predicate paths in a FuncGraph F.

The result of a control node is either used as part of the computation of a p-use argument of another control node or or is eventually used as part of the computation of the FuncGraph result. The former case is for an edge on a predicate path as defined above. The latter possibility represents a contribution to the result of the FuncGraph. We term this type of path a *value path* because the FuncGraph result is value-dependent on this path. Formally, a value path is any path in a FuncGraph that ends with the result edge of the FuncGraph and contains no p-use edges[7] We use the expression $ValPaths(F)$ to denote the set of all value paths in a FuncGraph F.

[7]Recall we have defined p-use edges to be only those argument edges to control nodes that predicate value selection, and not all the edges of a predicate path.

Figure 6 shows the value paths and predicate paths in the modified FuncGraph representation of PrefixEval in Figure 5. The thin solid lines represent the edges contained in the value paths of PrefixEval. The dashed lines represent the edges contained in the predicate paths of PrefixEval. Edges contained in both a value path and a predicate path are represented by thick solid lines.

The predicate paths and value paths of a FuncGraph represent all of the possible control and value dependency relationships present in a FuncGraph. Since every edge in a FuncGraph represents a data dependency, this means that every edge in a FuncGraph is contained in at least one predicate path or value path. Thus, the predicate paths and value paths together cover every edge in a FuncGraph.

The result of a FuncGraph can only depend on a value path. A particular value path may not always be selected by a control node. A group of value paths will all contribute to the result of a FuncGraph if they are each selected by the p-use argument edges of every control node through which they pass.

Two value paths will be used in the computation of one possible result value of a FuncGraph if they merge at a non-control node in the FuncGraph. Two edges e_1 and e_2 are said to merge if $NodeId(To(e_1)) = NodeId(To(e_2))$ and $From(e_1) \neq From(e_2)$. If the two edges merge at a non-control node then the result value of the non-control node is value-dependent on both edges. We say two paths merge if an edge on one path merges with an edge on the other path, and the two paths share a common non-empty path suffix beginning at the node where the two edges merge. We define the relationship between two paths that merge at a non-control node as a "sibling" relationship and we use the expression $Siblings(P1, P2)$ to denote that two paths $P1$ and $P2$ are siblings.

Figure 6: PrefixEval Predicate and Value Paths

We use the "sibling" relationship to define a set of value paths called a *value execution trace*, (denoted VET) which represents one possible computation of a FuncGraph result. Definition 1 formally defines a VET in a FuncGraph F.

Definition 1

A Value Execution Trace (VET) is a set of Value Paths $\mathcal{V}$ in a FuncGraph F such that $\mathcal{V} \subseteq ValPaths(F)$
$\wedge \forall p1, p2[(p1 \in \mathcal{V} \wedge p2 \in \mathcal{V} \wedge p1 \neq p2) \Rightarrow Siblings(p1, p2)]$
$\wedge \forall p[p \in \mathcal{V} \Rightarrow \neg\exists q[q \in ValPaths(F) \wedge q \notin \mathcal{V} \wedge Siblings(p, q)]]$.

Informally, a VET is a set of all sibling value paths. We use the expression $VETs(F)$ to represent the set of all VETs in a FuncGraph F.

The use of a VET to compute the result of the FuncGraph depends on the values of each p-use argument edge preceding any d-use argument edge of a control node that lies on any value path in the VET. The computation of the value on a p-use argument edge mirrors the computation of the result value of a FuncGraph. The value on the p-use argument edge is value-dependent on a set of sibling predicate paths which contain (and thus end on) the p-use argument edge. We refer to such a set as a *predicate execution trace*, (denoted PET). A PET represents the computation of one possible value of a p-use argument of a control node. Definition 2 formally defines a PET in a FuncGraph F.

Definition 2

A Predicate Execution Trace (PET) is a set of Predicate Paths $\mathcal{P}$ in a FuncGraph F such that $\mathcal{P} \subseteq PredPaths(F)$
$\wedge \forall p1, p2[(p1 \in \mathcal{P} \wedge p2 \in \mathcal{P} \wedge p1 \neq p2) \Rightarrow Siblings(p1, p2)]$
$\wedge \forall p[p \in \mathcal{P} \Rightarrow$
$\neg\exists q[q \in PredPaths(F) \wedge q \notin \mathcal{P} \wedge Siblings(p, q)]]$.

The expression $PETs(F)$ represents the set of all PETs in a FuncGraph F.

Each VET is directly control-dependent on the value of any PET whose p-use argument edge precedes a d-use argument edge of a member of the VET at a control node. The value of a PET may in turn be directly control-dependent on the value of another PET if the other PET's p-use argument edge precedes a d-use argument edge of a member of the first PET at a control node. We say a VET or PET $\mathcal{V}$ is directly control-dependent on a PET $\mathcal{P}$ if $\mathcal{P}$ is incident-on $\mathcal{V}$, denoted $IncidentOn(\mathcal{V}, \mathcal{P})$[8].

By transitivity, a VET $\mathcal{V}$ in a FuncGraph F is indirectly control-dependent on any PET $\mathcal{P}$ if $\mathcal{P}$ is incident-on a PET $\mathcal{P}'$ on which $\mathcal{V}$ is either directly or indirectly control-dependent. The predicate dependency set of a VET is the set of all PETs on which the VET is either directly or indirectly control-dependent. We denote this predicate dependency set of a VET $\mathcal{V}$ in a FuncGraph F by $PredDepSet(\mathcal{V}, F)$.

Recall that the value on a p-use argument edge of a control node is like the result value of a FuncGraph. There may be a choice of several different PETs that compute the p-use argument value. For any single execution of the function,

[8]It is assumed that both arguments are part of the same FuncGraph.

only one of these PETs is actually used to compute the value of the p-use argument. Any PETs that end on the same edge represent different possible computations of the value on that edge. We use the expression $EndEdge(\mathcal{P})$ to denote the edge on which a PET $\mathcal{P}$ ends.

A particular VET may be chosen by several different combinations of PETs in its predicate dependency set. Each of these different combinations represents one execution of the function which returns the value computed by the VET. Each VET paired with one of the combinations of PETs is termed an *execution trace*, (denoted ET). Definition 3 formally defines an execution trace of a FuncGraph F.

Definition 3

An Execution Trace (ET) in a FuncGraph F is an ordered pair $(\mathcal{V},\mathcal{S})$ such that $\mathcal{V} \in VETs(F) \wedge \mathcal{S} \subseteq PredDepSet(\mathcal{V}, F)$
$\wedge \forall \mathcal{P}[\mathcal{P} \in \mathcal{S} \Rightarrow (IncidentOn(\mathcal{P}, \mathcal{V})$
$\vee \exists \mathcal{P}'[\mathcal{P}' \in \mathcal{S} \wedge IncidentOn(\mathcal{P}, \mathcal{P}')])]$
$\wedge \forall \mathcal{P}[(\mathcal{P} \in PredDepSet(\mathcal{V}, F) \wedge IncidentOn(\mathcal{P}, \mathcal{V})) \Rightarrow$
$\exists! \mathcal{P}'[\mathcal{P}' \in \mathcal{S} \wedge EndEdge(\mathcal{P}) = EndEdge(\mathcal{P}')]]$
$\wedge \forall \mathcal{P}[\mathcal{P} \in \mathcal{S} \Rightarrow$
$\forall \mathcal{P}'[(\mathcal{P}' \in PredDepSet(\mathcal{V}, F) \wedge IncidentOn(\mathcal{P}', \mathcal{P})) \Rightarrow$
$\exists! \mathcal{P}''[\mathcal{P}'' \in \mathcal{S} \wedge EndEdge(\mathcal{P}') = EndEdge(\mathcal{P}'')]]]$

Figure 7 shows the VETs and PETs contained in the FuncGraph of the function PrefixEval. We use integers to label the VETs and PETs. Each edge is labeled by a parenthesized list of the VETs and by a bracketed list of the PETs in which the edge is contained. An edge may be contained in both VETs and PETs. In Table 1 we list the predicate dependency set for each VET in function PrefixEval and then list each execution trace in the function PrefixEval. The VET or PET number in the table corresponds to the label used in Figure 7.

A particular execution trace may actually be impossible to execute because of the nature of the operations used to compute the value of a PET. For example, ET4 in Table 1 could never be realized do to the nature of the and operator. In any case, the number of VETs in a FuncGraph is analogous to the number of acyclic paths in a flowgraph of a procedure containing compound predicate conditions. The magnified depiction of predicate dependency relations found

Table 1: Predicate Dependencies and Execution Traces

Predicate Dependency Sets

VET1	{PET1}
VET2	{PET1,PET2}
VET3	{PET1,PET2,PET3}
VET4	{PET1,PET2,PET3,PET4,PET5,PET6}
VET5	{PET1,PET2,PET3,PET4,PET5,PET6}

Execution Traces

ET1	VET1	{PET1}
ET2	VET2	{PET1,PET2}
ET3	VET3	{PET1,PET2,PET3}
ET4	VET4	{PET1,PET2,PET3,PET4,PET6}
ET5	VET4	{PET1,PET2,PET3,PET5,PET6}
ET6	VET5	{PET1,PET2,PET3,PET4,PET6}
ET7	VET5	{PET1,PET2,PET3,PET5,PET6}

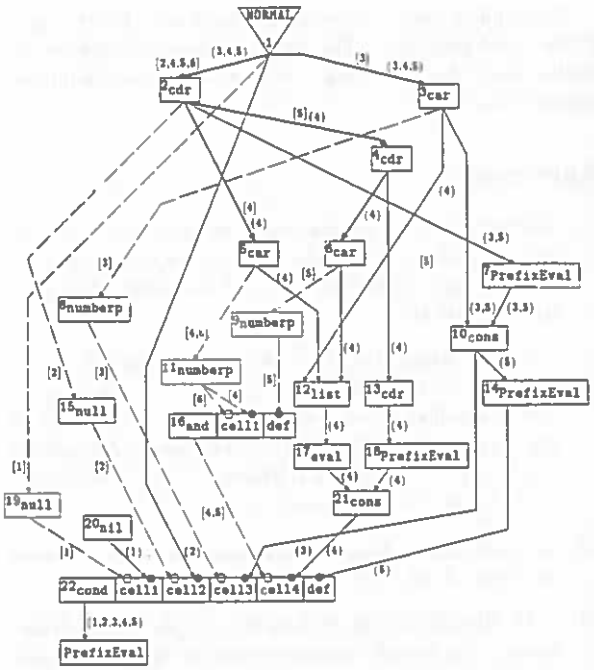

Figure 7: VETs and PETs of PrefixEval

in a FuncGraph gives us a finer count, namely the number of execution traces, which is analogous to the number of acyclic paths in a flowgraph of a procedure containing only simple predicate conditions.

7 Conclusions

The PET, VET, and ET structures of lispflows defined in the preceding Section serve as examples of the static analysis of functional programs supported by the lispflow model. The authors suggest that these structures can be used to define hierarchies of structural test criteria analogous to those of Rapps and Weyuker [12] and Ntafos [10]. These structures are also related to the notion of predicate scope [7] and predicate range [8]. The authors are currently developing other analytical properties of functional programs based on the lispflow model.

An issue that may be of greater importance is the appropriateness of the dataflow paradigm for modeling production dialects of LISP, and imperative features of programs generally. This is consistent with the work on data dependencies in imperative programs [4,12]. Also there are features of imperative programs which the flowgraph model does not handle. Software measures researchers have for some time been stymied by the inadequacy of measures based on flowgraphs to account for what goes on in expressions. "What goes on in expressions" is precisely what the lispflow model does handle. Additionally, flowgraphs are not particularly useful for modeling the interprocedural, or integration, structure of imperative programs. This is most apparent for recursively defined procedures. The authors are currently investigating a hierarchy of integration test criteria based on the lispflow model for Pure LISP.

The lispflow model is providing insight into the structure of functional programs. The dataflow paradigm may prove equally useful for describing a wide range of programming language features.

References

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

[2] Albert L. Baker, James M. Bieman, and Paul N. Clites. Implications for formal specifications: results of specifying a software engineering tool. In *Proceedings of the Eleventh Annual International Computer Software & Applications Conference (COMPSAC-87)*, IEEE Computer Society, Tokyo, Japan, October 1987.

[3] David M. Betz. Xlisp: an experimental object-oriented language. June 1986.

[4] J. M. Bieman and W. R. Edwards. Experimental evaluation of the data dependency graph for use in measuring software clarity. *Proc. 18th Hawaii International Conference on Systems Science*, 18:271-276, 1985.

[5] David L. Coleman and Albert L. Baker. *Lispflows: Modeling the Structure of Functional Programs*. Technical Report 88-11, Iowa State University, Department of Computer Science, 1988.

[6] J. B. Dennis. First version of a data flow procedure language. *Lecture Notes in Computer Science*, 19:362-376, 1974.

[7] W. A. Harrison and K. I. Magel. A complexity measure based on nesting level. *ACM Sigplan Notices*, 16(3):63-74, 1981.

[8] J. W. Howatt and A. L. Baker. Definition and design of a tool for program control structure measures. *Proc. COMPSAC85*, 214-220, 1985.

[9] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International Ltd, 1986.

[10] Simeon C. Ntafos. A comparison of some structural testing strategies. In *Proc. of the Nineteenth Annual Hawaii Int. Conf. on System Sciences*, 1986.

[11] Roger S. Pressman. *Software Engineering: A Practitioners Approach*. McGraw-Hill Book Company, 1987.

[12] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Trans. Software Engineering*, SE-11(4):367-375, 1985.

[13] Robert Wilensky. *LISPcraft*. W.W.Norton & Company, Inc., 1984.