LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

TYPE CHECKING IN GENERALIZED VAL

Computation Structures Group Memo 227
May 1983

Timothy Peacock

Thesis submitted to the Department of ElectricaL Engineering and Computer Science in partial fulfill- ment of the requirements for the S.B. degree.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Table of Contents

1 Introduction 2
- 1.1 Generalizing VAL 2
- 1.2 Type Checking Complexity 3
- 1.3 Synopsis 3

2 An Introduction to Generalized VAL 4
- 2.1 New Data Types 4
  - 2.1.1 Streams 4
  - 2.1.2 Functions 5
- 2.2 Other Changes 6
  - 2.2.1 No FOR or ITER Constructs 6
  - 2.2.2 Recursive Types 7
  - 2.2.3 Partial Record Types 8
- 2.3 Optional Declaration of Types 9
  - 2.3.1 Ease of Use 9
  - 2.3.2 Generic Functions 9
  - 2.3.3 Type Checking Implications 10

3 Type Checking Problems 10

4 Development of Algorithm 12
- 4.1 Data Type Graphs 12
  - 4.1.1 Operator Nodes 13
  - 4.1.2 Type Nodes 13
  - 4.1.3 Constraints 14
- 4.2 Goals 15
  - 4.2.1 Type Correct 16
  - 4.2.2 Consistency 16
- 4.3 Relative Constraints 17
  - 4.3.1 Merge 20
  - 4.3.2 Type Determination 21
  - 4.3.3 Link Time Information 22
- 4.4 General Algorithm 23

5 Constraint Algorithms 24
- 5.1 Action Routines 24
  - 5.1.1 SetType 25
  - 5.1.2 SetMultType 26
  - 5.1.3 Merge 27
  - 5.1.4 Equiv 31
  - 5.1.5 IsRecursive 32
  - 5.1.6 TypeEqual 34

6 Examples 39
- 6.1 Example 1 39
- 6.2 Example 2 44

6.3 Example 3 49
7 Module Link Algorithm 53
7.1 Type information available 53
7.2 Binding 54
7.2.1 Assertion 54
7.3 Example 55
8 Producing Data Type Graphs 56
8.1 Data Flow Graphs 56
8.2 Data Type Graphs 57
9 Conclusion 57
9.1 Alternatives 58
9.2 Applications to Other Languages 58
9.3 Further Work 58

Appendix A: VAL Operators and Constructs **59**

10 Basic Operators 59
10.1 Error Tests 59
10.2 Equal and Not Equal 59
10.3 Boolean operations 60
10.4 Type Conversion Operations 61
10.5 Real and Integer Operations 61
10.6 The empty operation 63
10.7 Array Operations 64
10.8 Stream Operations 67
10.9 Record operations 68
10.10 Union Types 69
10.11 Constants 70
11 Basic Constructs 70
11.1 If Then Else constructs 70
11.2 Tagcase constructs 72
11.3 Forall constructs 74
12 Functions 75
13 Special Nodes 76

TYPE CHECKING IN GENERALIZED VAL

by

TIMOTHY MARK PEACOCK

Submitted to the Department of Electrical Engineering and
Computer Science on May 27, 1983 in partial fulfillment
of the requirements for the degree of Bachelor of Science
in Computer Science and Engineering.

ABSTRACT

VAL, in its current implementation, is a strongly typed language. That is, all of the variables used in a VAL program must have their types explicitly declared by the programmer. This provision ,while helping to simplify compile time type checking, can become quite bothersome to the programmer. Therefore, all type declarations will be optional in the proposed revision of VAL. This capability will allow the programmer to write generic functions (functions that perform analogous operations on different data types).

The immediate problem with compiling revised VAL is how to type check VAL modules with no type declarations. Our approach is to represent each VAL module as a data type graph. Type values filter through such a graph, passing through various operator nodes. Examination of each operator node places constraints on the type values. If every operator is examined and none of the constraints are in conflict, the module is type checked. My thesis is that an algorithm can be devised to do all type checking in a module at compile time, and that type determination, a related problem, can be accomplished at link time.

Thesis Supervisor : Jack B. Dennis

Title: Professor of Computer Science and Engineering

Acknowledgments

```
"Well I pick up my guitar and play,
   Just like yesterday,
 and I get on my knees and pray,
  we don't get fooled again."
                - Pete Townshend
```

I wish to express my gratitude to all the members of the Computation Structures Group for their help with this thesis. Notably Andy Boughton and Bhaskar Guharoy for their help with system related problems, and Bill Ackerman for his help with VAL itself and the VAL compiler.

I would like to thank my thesis advisor Prof. J. Dennis for having a thesis topic available when I stumbled into his office in January, for his help in understanding the problems inherent in the thesis, and for his constructive criticisms of the actual thesis document. In addition, I feel I should mention Prof. W. B. Watson, whose history courses kept me interested in staying at MIT when I might otherwise have left.

I also would like to thank the brothers of Xi chapter Tau Epsilon Phi fraternity for providing a sane environment in which to live at MIT; my brother, for not describing what its like here before I came; and my parents, for their support both monetary and physical.

```
"Sometimes the lights all shining on me,
    Other times I can barely see,
       Lately it occurs to me,
  What a long strange trip it's been."
               - Robert Hunter
```

Acknowledgments

"Well I pick up my guitar and play,
Just like yesterday,
and I get on my knees and pray,
we don't get fooled again."
- Pete Townshend

I wish to express my gratitude to all the members of the Computation Structures Group for their help with this thesis. Notably Andy Boughton and Bhaskar Guharoy for their help with system related problems, and Bill Ackerman for his help with VAL itself and the VAL compiler.

I would like to thank my thesis advisor Prof. J. Dennis for having a thesis topic available when I stumbled into his office in January, for his help in understanding the problems inherent in the thesis, and for his constructive criticisms of the actual thesis document. In addition, I feel I should mention Prof. W. B. Watson, whose history courses kept me interested in staying at MIT when I might otherwise have left.

I also would like to thank the brothers of Xi chapter Tau Epsilon Phi fraternity for providing a sane environment in which to live at MIT; my brother, for not describing what its like here before I came; and my parents, for their support both monetary and physical.

"Sometimes the lights all shining on me,
Other times I can barely see,
Lately it occurs to me,
What a long strange trip it's been."
- Robert Hunter

1 Introduction

This thesis is based on work done by the M. I. T. Computation Structures Group. The group is applying the concepts of functional programming and data flow computer architecture to the design of general purpose computer systems. The goals of the group are

> to present a system model for a kind of ideal multiprogrammed computer system, one that would serve many users in a way permitting sharing of the products of their individual programming efforts consonant with the principles of program modularity -- the ability to build program units which can be combined to form higher units, etc.

[Dennis 81]. This goal was first articulated inside the group back in 1966 in [Dennis 66]. Despite the amount of time since, the complete goals of that original paper have not been achieved in a practical computer system. The group is now working on a *Base Language* that can be used to specify a general class of computer systems.

VAL, a Value-Oriented Algorithmic Language, is intended to be the base language for a computer system. Here, a computer system is the combination of hardware, firmware and software that realizes the base language. The base programming language must be sufficiently complete so as to provide the user access to all essential features of a computer system. A programmer should not have to depart from the language in order to express any requirement of an application. VAL, as described in [Ack 79], because of its coherent design, can support large portions of all computer applications, as argued in [Dennis 81].

1.1 Generalizing VAL

VAL, when originally specified, was intended to be a base language for a static data flow architecture. As such, it was decided to make VAL a strongly typed language, that is a language in which all data types had to be explicitly declared. In addition, in order to limit the scope of the problem for implementation, two data types were omitted, the stream and the function. Now, an updated version of VAL is being implemented. The new version will make all type declarations optional. This allows the programmer to write generic functions, as well as eliminating the need for repetitive declarations. The stream type is included to provide for inter-process communication. The function type is added to allow functions to be passed as parameters to other functions.

1.2 Type Checking Complexity

The new version of VAL requires a much more complex type checking algorithm than the old version. This is primarily due to the removal of type declaration requirements. Formerly, the VAL compiler was able to do type checking by using a technique for determining equivalent states of finite state automata by separating them into equivalence classes. The technique has been used to find the equivalence of modes in Algol 68 [Zosel 71, Kral 73]. Unfortunately, this algorithm depends on the presence of all type information to separate types into equivalence classes. In the new version of VAL, it is often necessary to determine type equivalence before all type information is available. This introduces the notion of type consistency. Two types are consistent if their basic types are not contradictory and none of their sub-types are inconsistent. Two contradictory basic types are 'real' and 'char'. Two non-contradictory basic types are 'any' and 'stream'. Thus, the new version of VAL must be type checked not by achieving for type equivalence, but rather by achieving type consistency.

Type consistency is hard to determine if the two types are recursive. Recursive types, data types that point to themselves, are legal in VAL provided they include at least one union type. There is no simple algorithm for determining the consistency of recursive types. What algorithms there are, are based on examining the structure of the tree which represents the type in some exhaustive manner.

Type information can be gathered by representing the VAL modules as data type graphs. The translation of a module to its graph form is straightforward. By examining each operator node in the type graph, constraints are placed on the types. Often, types are constrained to be a single type, such as integer. If the entire graph is examined, and none of the constraints are in conflict, then the module is type consistent, and thus, type checked.

1.3 Synopsis

After presenting the differences between generalized VAL and the old version, this thesis examines the problems inherent in type checking the new VAL. An algorithm to type check generalized VAL is then presented. The algorithm allows intra-module type checking to be accomplished at compile time. Inter-module type checking, and module type determination are completed at module link time. After presenting the algorithm in general, a major portion of the algorithm is examined in detail. The purpose of the detailed examination is to show that

the algorithm can be implemented. Next, a few examples of the algorithm are presented to aid comprehension. Following the examples, the module link time algorithm is presented. Before concluding, some notes on producing data type graphs are offered.

2 An Introduction to Generalized VAL

VAL was originally designed as a base language for a static data flow architecture. It was also designed to support multi-processing and multi-programming. The enhanced version of VAL is intended to extend this support while at the same time making VAL functions easier to write.

The new version of VAL has three new data types. These are the stream type, the function type and the empty type. The stream type is used for inter-process communication. The function type allows functions to be passed as arguments to other functions. The empty type is a single type used to represent either an empty array or an empty stream.

The new VAL has no FOR or ITER constructs. These constructs are unnecessary as recursion serves the same function. The new version of VAL allows recursive data types, but only if the type contains a union type. This restriction provides a defense against infinitely recursive types. The new VAL also allows record types to be initialized in parts.

Type declarations will be optional. This eliminates the requirement that all data types be explicitly declared by the programmer. Optional type declarations will provide the ability to write generic functions.

2.1 New Data Types

2.1.1 Streams

The new version of VAL will have a stream data type in order to express computations that are often expressed as coroutines or as sets of cooperating processes. Coroutines or cooperating processes have been used for two main reasons. One, they alleviate the tendency of large computations to produce large intermediate data structures. Two, they allow subcomputations to be executed concurrently [Weng 80].

VAL will use streams instead of coroutines or synchronization primitives because experience has shown that writing applications with coroutines or the primitives is quite difficult. The correctness of such programs is hard to establish. This leads to programming errors resulting in deadlocks or unwanted nondeterminacy. The VAL version of streams should save the inherent concurrency while allowing the computation to be determinate and free of deadlocks. Also, streams support program modularity in the following sense: the overall behavior of modules can be expressed as a function of streams and characterized using denotational semantics [Weng 80].

The new stream data type will have the following operations:

operation	notation	functionality
create	stream(V1,...,Vn), n >= 1	T1,...,Tn -> stream[T]
is stream empty	null[stream[T]]	stream[T] -> bool
obtain first element	first(V)	stream[T] -> T
obtain rest of stream	rest(V)	stream[T] -> stream[T]
prefix elmt to stream	affix(V1,V2)	T,stream[T] -> stream[T]

The stream data type introduces almost no additional complexity to the type checker. Indeed, streams can be type-checked in the same manner as arrays.

2.1.2 Functions

The second new data type is the function. Functions are currently the highest level of program structure in VAL. With their inclusion as data types, it will be possible to pass functions as parameters to other functions.

One good use for function parameters is the ability to write generalized functions. For example, one might write a generalized summation function. Consider the function SumSquares shown below.

```
function SumSquares (a,b:integer returns integer)
        for x : integer := a; y : integer := 0;
        do if x <= b then iter y := x*x+y; x := x+1; enditer
           else y
           endif
        endfor
```

```
endfun
```

This function will sum the squares of a range of integers from a to b. However, if then one wanted to sum the cubes of a range of integers or perform some other summation, it would be necessary to write a whole new summation function. A better solution would be to write the general summation function listed below.

```
function summation (a,b : integer ;FofX, step :
                            function returns integer)
  for x : integer := a; y : integer := 0;
  do if x <= b then iter y := FofX (x) + y;
                         x := step (x); enditer
     else y
     end if
  endfor
endfun
```

Here, FofX is the function of x to be summed and step is the increment used between a and b. Assuming that the functions square, cube and plus1 exist, the following calls on summation would suffice to sum the squares and the cubes of a range of integers from a to b.

```
summation ( a, b, square, plus1 )
summation ( a, b, cube, plus1 )
```

This is just one example where function data types allow more flexible functions to be written. Another might be in writing parsers for which input may come from different sources.

The type checking of functions is fairly complex. The type checker must make sure that a function is not passed as a parameter to itself. This would be an unending or infinite recursive type. Determining equivalence of function type is also complex. The complexity is inherent in the need to check two sets of sub-types; the argument sub-types, and the result sub-types.

2.2 Other Changes

2.2.1 No FOR or ITER Constructs

The FOR construct, and its related ITER construct, are not in the new version of VAL. This is because they provide no ability to VAL not provided by calls on recursive functions. In order to better see this, consider the conversion of the general FOR construct below to a recursive

implementation. (Only 1 example of a FOR body is shown but all can be easily converted.)

```
FUNCTION foo()
        FOR exp1;
        DO IF exp2 THEN ITER exp3 ENDITER
                   ELSE exp4
           ENDIF
        ENDFOR
ENDFUN
```

Where exp1 - exp4 are arbitrary VAL expressions. This converts into the following recursive program.

```
FUNCTION foo (ArgList)
        LET args := ArgList      ; args are list of vars in exps
        IN IF exp2 THEN foo (exp3)
                   ELSE exp4
           ENDIF
        ENDLET
ENDFUN

foo (exp1)
```

The absence of the FOR and ITER constructs eases the type-checker's task. The iter construct was the sole source of feedback loops in a data flow graph. The elimination of such feedback simplifies the type checking algorithm.

2.2.2 Recursive Types

VAL's new type system will allow partially recursive, and mutually recursive types. However, infinitely recursive types are illegal. Here is an example of a infinitely recursive type:

```
type T = array [T];
```

A quick analysis of this data type reveals it to be an infinitely dimensioned array. Thus, it can have no physical significance, indeed it can not even be initialized on a computer. Here is an example of a partially recursive type:

```
type stack = oneof[empty:null;
                  top:record[value:integer;rest:stack]];
```

A stack type is clearly physically significant. Furthermore, this type can be implemented because the union allows the infinite recursion to be broken. Here is an example of a mutually

recursive type:

```
type stack = oneof[empty:null;top:element];
type element = record [value:integer;rest:stack];
```

This is just an alternate representation of a stack. The union type again helps to prevent infinite recursion. The difference is that the two types may be defined in two different modules. Mutual recursion of data types between modules is permitted.

The key difference between legal and illegal recursive data types, is the presence of some structure that guards against infinite recursion. In VAL, the only such structure is the union type. This type does not necessarily provide complete protection (one can still program the infinite recursion). However, if it is not present, the recursive data type is inherently infinite, and thus illegal.

Recursive types introduce great complexity into the type checker. Recognizing them is straightforward. However, since recursive union types are legal in VAL, it becomes necessary to determine if two recursive types are consistent or inconsistent. The basis for solving this problem is the complete examination of the trees which represents the types. Another problem is that mutually recursive types that recurse between modules can only be recognized at module link time.

2.2.3 Partial Record Types

Currently, when one wants to create a record type, one must specify all the elements of the record type. With complicated record types this can be an unnecessary hassle as the function may only use part of the record type. For instance:

```
type employ = record [name:array[char];id:int;pay:real;sex:char];
```

The employ type may be used to keep track of an employee's work status. However, a payroll program would not care what sex the employee is, just his or her salary. Thus it would like to be able to refer to the type as shown below.

```
record [name:array[char];id:int;par:real]
```

Other, more extreme examples can be found. This partial specification of a record type is

known as a partial record type. The type has three main advantages; one, it is easier for the programmer to use; two, it can be used to improve security; and three it helps to promote data format independence. (For the last two advantages, it is similar to a sub-schema of a database.)

2.3 Optional Declaration of Types

Currently, every VAL variable has its type explicitly declared, be it a parameter of a function, a loop value or whatever. The new version of VAL will make such type specification optional for all variables. That means that, if the programmer so chooses, no values need be given explicit type declarations at all.

2.3.1 Ease of Use

Consider the gain shown in the following example which begins by setting x equal to y.

```
function equate ( y : record [a : integer; b : real; c : array[int]
                     returns T)
        let x : record [a : integer; b : real; c : array[int]]
            x := y;
```

With no type declarations, the same function fragment would be :

```
function equate ( y )
        let x := y;
```

In some ways, this is clearly superior. Of course, the programmer will have to specify some types for his or her own sake. Otherwise, the ability to use abstract data types will be lost, to say nothing of program clarity.

2.3.2 Generic Functions

Another useful result from making type specification optional is the ability to write generic functions. A generic function is one that performs the same or analogous operation on its parameters regardless of their data type. For example, a generic add would add two integers, add two reals, or perhaps vector add two arrays.

A more powerful example of generic functions might be in the implementation of a stack. The

following example would not be legal in the current implementation of VAL.

```
function push (val,stack)                       ; push value onto stack
        make record [top : record [value:val;rest:stack]] ; rtn stack
endfun

function pop (stack)                            ; return a value from a stack
        tagcase stack                           ; and new stack
        tag empty : null
        otherwise : stack.value,stack.rest
        endtag
endfun
```

Here the stack type is the same as that shown in the recursive types section with the exception that 'value' can be any legal VAL type.

2.3.3 Type Checking Implications

The obvious problem with eliminating type specifications is how to determine a value's type in order to type check it. For the basic VAL types, integer, real, null & bool, this is easily done by inspection of the expressions to which they are set. However, if the value is a stream, array, record or union type, problems can arise. Consider the case where a union type is recursive. Unless the type is specified, deducing the actual structure may prove tricky. Even worse, if the recursion goes across module boundaries, the type might not even be known until linkage time.

3 Type Checking Problems

Type checking the new version of VAL is a far from trivial problem. By type checking, I mean checking to see if all the data values used in a VAL module have the correct data type for their use. For example, an integer value can not be added to a boolean value. The main problem with type checking in new VAL is that all type declarations are optional. In the worst case, this means that none of the values used in a module will have any type information available at the time when type checking should commence. The type checking problem then becomes one of type determination.

The key to type determination in VAL is examining the constraints that VAL operators place

on their arguments. For example, the AND operator constrains its two arguments to be boolean types. It also constrains its result to be boolean. While it would be possible to examine these constraints directly from the textual representation of a VAL module, I feel that a data type graph representation is more effective. Once into the graph representation, all data types can be determined in a straightforward manner by an exhaustive examination of all operator nodes, except recursive data types.

The determination of recursive data types is tricky. While an existing recursive type can be easily recognized, type checking depends on the ability to recognize the equivalence of data types. Determining the equivalence of recursive types, when all type information is available, is possible. Indeed, the old version of VAL has such a capability in its compiler. Unfortunately, what is needed here is to determine equivalence of recursive types from incomplete information.

This problem leads to the removing of the requirement for type equivalence. Instead, what is required is type consistency. Consistency of recursive types can be determined in a brute force manner by repetitive examination of the recursive types tree representation. The ability to determine type consistency from incomplete information allows new VAL to be type checked in one pass. A VAL module is type checked if, at the end of the examination of data type graph operator nodes, no inconsistent types were found.

However, this does not mean that all the values in the module have had their type determined. A value is said to be type determined if and only if its basic type is known and the basic types of all its sub-types are known. (A basic type is one of {null, integer, real, char, bool, empty, array, stream, record, oneof, function}. Note that only array, stream, record, oneof and function types have sub-types.)

A VAL module can not be run until it is type determined. It is possible that the module can not be type determined until module link time. For example, a call on function *add*, add (1.2, 1.3), would provide the information that both of add's arguments are of type real. Yet such a call could legitimately be in another module. The problem is to provide the ability to type determine VAL modules at link time, without the need to re-examine their data type graphs.

One solution to this problem is to provide a symbol table for each module. The symbol table would provide pointers to basic types which would point to sub-types and so on. Comparison

of the two symbol tables (remember that functions, as data types, are in the symbol table) will provide the necessary information in a straightforward manner.

With this approach in mind, the development of a correct algorithm may proceed.

4 Development of Algorithm

This section concerns itself with the purpose and development of the algorithm. It also presents alternate approaches to accomplishing the algorithm's goals. The algorithm's basic goal is to type check the new version of VAL. The normal definition of 'type checking' is that a program is type checked if and only if, 'iff', all operations in the program can be executed for the arguments specified by the program. This definition is not used by this algorithm, as it is a run-time definition. The algorithm wishes to show that a modules can be type checked at compile time. Consequently a compile time definition will be presented. First, however, a few basic definitions are needed.

4.1 Data Type Graphs

The objects on which the algorithm operates are data type graphs. These objects are derived from data flow graphs as presented by [Dennis 75].

A data type graph is a directed bipartite, acyclic graph. The two types of nodes in the graph are operator nodes, and type nodes. Directed means that the links between nodes can only be traversed in one direction. Bipartite means that the two types of nodes must alternate in the graph. Acyclic means that if one starts at an arbitrary node, by following the directed links, one can never return to the original node.

A data type graph can be thought of as two sets, the set of operator nodes and the set of type nodes. Each link is a pair (n1, n2) where one of n1, n2 is a type node, and the other is an operator node.

4.1.1 Operator Nodes

The set of operator nodes is defined as follows :

1) An operator node is oneof { + , -, *, A-sel, = , is_undef, ... } (for a complete listing of operator nodes, see the Appendix). Note that the set of operator nodes is directly related to the set of legal VAL operations and constructs. All VAL operators translate directly into operator nodes. VAL constructs can be translated into graph structures built up from operator nodes. Again see the Appendix for further details.

2) Associated with each operator node is a set of type constraints. These constraints apply to the type nodes which are the operators arguments and results. A type node is the argument of an operator node, iff one can traverse a link from the type node to the operator. A type node is the result of an operator node, iff one can traverse a link from the operator node to the type node. Remember that this is a directed graph. Note, if one visualizes the graph as beginning at the top and being directed towards the bottom, then the operator node's arguments are 'above' it and the results are 'below' it. Constraints are defined after type nodes.

4.1.2 Type Nodes

The set of type node is defined as follows:

1) A type node has a type number. Note that in the set of type nodes in any specific data type graph, no element may have the same type number as another.

2) Each type node has an associated type specification. A type specification contains three parts. They are as follows:

2a) A basic type set. This a set with a maximum of four elements. Each element is a member of the set {any, null, integer, bool, real, char, empty, array, stream, record, oneof, function}. In short, all the data types allowed in VAL with the inclusion of the type 'any', which means the type could be any of the others.

2b) A sub-field list. This is a list of character strings which are ordered alphabetically. This list is empty except when the basic type set has the element record or oneof in it. In that case,

this list is the field names of the record or oneof's sub-types.

2c) A pair of sub-type lists. A sub-type list is a list of pointers to type nodes. The second list is always empty, unless the basic type set contains the element function. The first list is empty, unless the basic type set contains the element array, stream, record, oneof, or function. Note that cyclic type specifications are possible and legal. A cyclic type specification is known as a 'recursive' type specification. Also note that the order of the sub-type lists is important. In the case of the basic type set = {record} or {oneof}, corresponding entries in the first sub-type list and the sub-field list are matched. That is to say, the first field in the sub-field list is the field name of the first type node in the sub-type list.

3) Each type node has associated with it an equivalence set. An equivalence set is a set of pointers to type nodes. The meaning of an equivalence set is that the type node has been constrained to be equivalent to all the type nodes in its associated equivalence set.

There are two special classes of type nodes, formal parameters and free variables.

A formal parameter is a type node which is an argument node of the operator node with the label 'PROCED'. This operator node is the node used to represent a function header. Thus a formal parameter is a type node representing an argument to a VAL function.

A free variable is any type node in the graph that is not the result node of any operator node in the graph and is not a formal parameter. VAL's syntax limits free variables to being type nodes with the basic type set equal to {function}. That is, VAL free variables are just calls on functions defined externally to the VAL module.

4.1.3 Constraints

As was mentioned previously, an operator node has a set of constraints that apply to its argument and result type nodes. The set of possible constraints are defined as follows:

1) A constraint may be that one type node must be equivalent to another. This means that the associated type specifications of the two type nodes must be *type consistent* (see below) and that the two type specifications should be *merged* (also see below).

2) A constraint may be that the type specification of a type node must:

2a) Have a basic type set of one element and that element is equal to oneof (null, real, bool, char, integer, empty).

2b) Have a basic type set of multiple elements, all of which are members of (real, integer, bool, char).

2c) Be equal to some type specification with a basic type set with the element = oneof (array, stream, record, oneof, function).

4.2 Goals

As was stated previously, the goal of the algorithm is to show that a program can be type checked by compile time. What was also stated is that the algorithm operates on data type graphs. An assumption of this thesis is that any VAL program can be mapped into a data type graph in a manner that preserves the functionality of the program. This assumption is valid as it has been demonstrated to be true for data flow graphs [Weng 80], and the only difference in graph construction would be the insertion of type nodes between all nodes in a data flow graph, a trivial process. (For an introduction into the production of data type graphs from VAL modules, see Section 8.)

The goal of the algorithm is now to insure that a data type graph is type checked by compile time, or to insure that a data type graph is *type correct*. The algorithm operates on a data type graph by examining each operator node in the set of operator nodes in the graph. For each operator nodes, the associated constraints are applied to the argument and result type nodes. The result of the algorithm is a new data type graph.

Formally, if the algorithm is a function, *check*, with G and G^* being data type graphs, then:

```
G* := check (G)
```

If the algorithm attempts to apply a constraint to a type node, and that constraint is not consistent with that type node's previous constraints, then an error is signalled. The assertion of this thesis is that if G^* is produced, and no error is signalled, then the data type graph, G^* is type correct.

4.2.1 Type Correct

An arbitrary data type graph G^* is type correct if and only if every operator node in G^* is *type satisfied*. An operator node is type satisfied if and only if the constraints associated with that node are consistent with the argument and result type nodes of that operator node. For example:

If the operator node was an *add* node. The constraints would be; 1, all argument and result type nodes are equivalent; and 2, each argument and result type node should have the basic type set {real, integer}. If the two argument and the one result type node all had the basic type set {real}, they would be **consistent** with the constraints, and thus the node would be type satisfied.

4.2.2 Consistency

Several times in this section the notion of consistency has been mentioned. Usually, it was mentioned in terms of two type nodes being type consistent with each other. The other case was when a constraint was mentioned as being consistent with a type node. The latter is just a special case of the other. If you envision the constraint as just a type node, T_c, embodying the information in the constraint (e. g. T_c -> basic type set = {real}), then a constraint is consistent with the type node if T_c is consistent with the type node.

Two type nodes are type consistent if and only if their type specifications are consistent.

Two type specifications are consistent if and only if:

1) Their basic type sets are consistent.

2) Their sub-type lists are consistent.

Two basic type sets are consistent if:

1) One of the basic type sets contains the element 'any'. OR

2) The intersection of the two sets is not empty. OR

3) One of the sets contains the element 'empty', and the other contains either 'array' or

'stream'.

Two sub-type lists are consistent if and only if:

1) The sub-field list is empty, AND the corresponding type nodes in each of the sub-type lists are consistent. OR

2) For each case where an element in one sub-field list is equal to an element in the other, the corresponding elements in the sub-type lists are consistent. If no sub-fields match, the sub-type lists are consistent.

4.3 Relative Constraints

The second assertion of this thesis is that the data type graph produced by the algorithm, G^*, is more or as constrained as the original graph, G. A data type graph, G^*, is *more constrained* than a graph, G, if:

1) None of the type nodes in G^* are less constrained than the type node with the same type number in G.

2) At least one of the type nodes in G^* is more constrained than the type node with the same type number in G.

A type node, T_1, is more constrained than a type node, T_2 if:

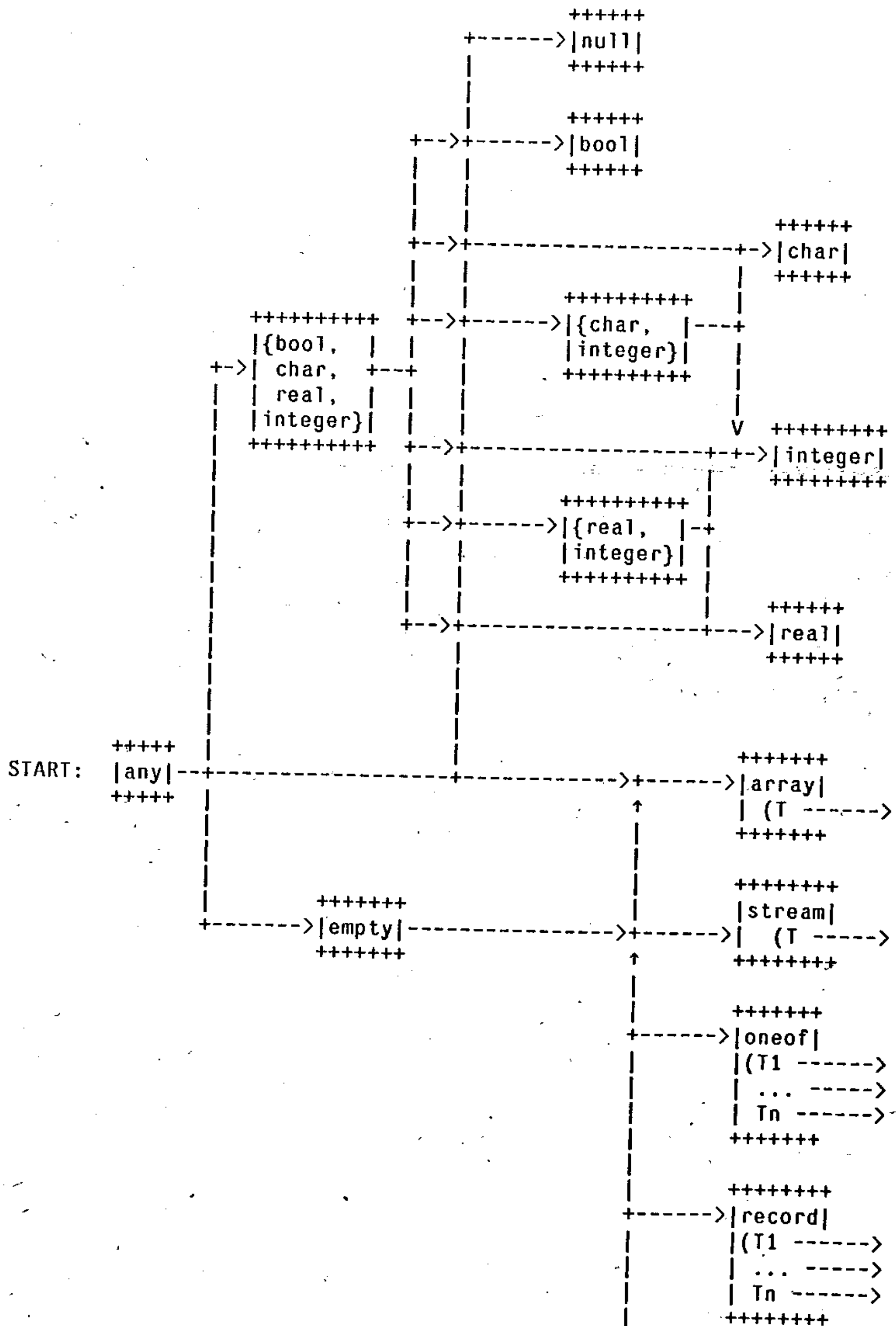
1) The equivalence list associated with T_1 has more entries than the list associated with T_2. OR

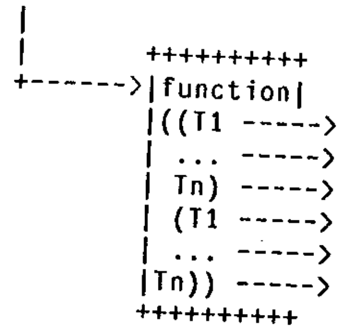
2) The type specification associated with T_1 is more constrained than the specification associated with T_2.

Before examining the concept of 'more constrained', as it applies to type specifications, note two facts. One, because T_1 and T_2 have the same type number, and the algorithm will **never** delete an element from an equivalence list, the equivalence list for T_1 in G^* will always have at least as many elements as the list for T_2 in G. Two, because the two type nodes have the same type number an no constraint may be applied to T_2 that is inconsistent with its existing

constraints, then T_1 must be consistent with T_2.

In the diagram that follows, the farther to the right that a type specification gets, the more constrained it is. Note that if a type specification reaches a point where it is unable to go further to the right, it is constrained as far as possible. Indeed, the type of that node is said to be *determined*. The diagram itself represents a sort of graph with directed links from left to right. Note that the graph is cyclic.


null
bool
char
{char, integer}
{bool, char, real, integer}
integer
{real, integer}
real
START:
any
array
(T
stream
(T
empty
oneof
(T1
...
Tn
record
(T1
...
Tn



Where all the links at the right of the diagram point to the label 'START:' on the extreme left of the diagram. The links that come out of the nodes labeled array, stream, oneof, record and function, are links to the type specifications of the associated sub-types in the sub-type lists.

A way of visualizing the algorithm is to picture all type nodes starting off with the type specification 'any'. That is, they all start off at the extreme left of the diagram. As the algorithm works, more and more constraints are placed on each type spec. Thus, each type spec moves towards the right of the graph. If a type spec reaches a node where there is no link to the right, that type spec is said to be determined. It is impossible for the algorithm to make the type spec go back towards the left of the graph, i. e. to remove a constraint. This is because the algorithm always merges constraints, and the definition of merge, precludes such removal.

4.3.1 Merge

A merge is a function that takes in two type specifications and sets each of them equal to the union of the two. If the two specs are not consistent, merge will signal an error. Note that union is used here to mean the union of the information provided, not a simple union of the associated sets and lists.

The merge of T_{s1} and T_{s2} where T_{s1} and T_{s2} are type specifications is defined as:

1) If T_{s1} and T_{s2} are inconsistent, signal an error.

2) The basic type set is the intersection of the two basic type sets.

3) If the two sub-field lists do not have the same arity. The sub-field list is the union of the

sub-field list. Merge is called recursively on the elements of the two sub-type lists that have equal field names. Those elements with no corresponding type in the other spec are just inserted into the correct position in the new sub-type list. Otherwise, the merged type node (or rather, the type node that points to the merged type spec) is inserted into the position.

4) If the sub-field lists have the same arity, the sub-field list is just one of the sub-field lists. The sub-type list is just one of the sub-type lists, after merge has been called for all the sub-types in corresponding positions in the two sub-type lists.

Note that the definition of merge contains a very important property. Namely, a merge **never** removes a constraint from a type specification. Thus, an operator node's type constraints, once applied, are always preserved.

It is this property that allows the second assertion to be made. The property ensures that all type specifications always end up as or more constrained than they were initially. Thus, the graph G^* is as or more constrained than G.

4.3.2 Type Determination

Type determined is a term that applies to VAL modules meaning that the types of all the data values are known, at least enough to execute the program. A data type graph is said to be type determined when:

All the formal parameters and free variables in the data type graph are type determined.

Thus, a data type graph is type determined if the type nodes which represent its formal parameters and free variables are type determined. A type node is type determined if and only if:

1) Its basic type set has one element and that element is not 'any'.

2) All of its sub-types are type determined.

This thesis asserts that, if it is possible to produce a graph G^* which is type determined, starting from a graph G, then this algorithm will do so. Another way of phrasing that assertion is, that G^* is the most constrained version of G possible from the information in G.

More formally:

1) In G^*, there is no operator node for which the algorithm defines a further constraint of a type specification.

2) Any order of examination of operator nodes would lead to the same graph G^*.

The reasoning is as follows:

1) The only constraining information in a data type graph is the constraints derived from the operator nodes.

2) All constraints derived from operator nodes are applied to the type nodes.

3) No constraint is applied in such a way that it destroys an existing constraint.

4) The presence of equivalence lists allows constraints to be propagated around a data type graph.

5) Therefore, all constraints are applied to all type nodes.

6) Therefore, G^* is the most constrained version of G possible.

4.3.3 Link Time Information

A VAL module can not be executed if it is not type determined. Yet, the algorithm will often finish when the data type graph is not type determined. This is because references to external functions often lead to non-determined type nodes. This forces the algorithm to allow for type checking at module link time. The algorithm saves the following information in order to be able to perform type checking at link time:

1) A symbol table with the identifiers of the formal parameters and free variables. Associated with each identifier is a pointer to the type node which represents that parameter or variable. If the entry in the table is type determined, a flag is set.

2) The type specification and equivalence list associated with each type node referenced in the symbol table, and associated with any type nodes referenced by the specifications or

equivalence lists.

This information is used by the link time algorithm as specified in section 7.

4.4 General Algorithm

Here is a list of steps executed by the type checking algorithm. Remember, the input to the algorithm is a data type graph 'G'.

1) For each operator node in the graph, apply the associated type constraints to the node's argument and result types. Signal any errors.

2) When done with step 1. If any error's signalled, the module is not type correct. Resignal errors and end.

3) Otherwise, examine the symbol table to see if the module is type determined.

4) If so, delete all nodes in graph not referenced by symbol table, directly or indirectly. End.

5) Else, just delete the operator nodes and end.

The assertion of this thesis is that if G^* is produced and no error is signalled (the algorithm above reaches step 4), then the module is type correct.

The reasoning is as follows:

1) All operator node associated constraints were applied. This is implicit in the algorithm.

2) No constraints were applied that were not consistent with another constraint. This follows from no error being signalled.

3) Therefore, all operator constraints are consistent with the operator's argument and result types. This follows from all constraints being applied.

4) Therefore, all operator nodes are type satisfied. This follows from the definition of type satisfied.

5) Therefore, the data type graph is type correct.

5 Constraint Algorithms

It is now necessary to show that the preceding algorithm can be implemented. The translation of VAL modules into data type graphs is straightforward and is summarized in Section 8. Algorithms for searching acyclic graphs are well known. I suggest that, for efficiency reasons, the graph be searched in a bottom-up, breadth-first manner (see Section 5.1.6). What is neither straightforward nor well known is how to apply type constraints. Accordingly, this section is devoted to a precise algorithm for applying type constraints.

5.1 Action Routines

Type constraints are embodied in action routine calls. When an operator node is examined, its associated constraints are found by performing a large case instruction on the operator node's identifier. The constraints are embodied in references to action routines. The execution of these routines applies the constraints. For a complete listing of operator node's and their associated constraints, see the Appendix.

There are four action routines. These are *SetType*, which corresponds to constraint 2a in Section 4.1.3; *SetMultType*, 2b; *merge*, 2c; and *equiv*, 1.

Here is the information that is available in the type node: the type number, the basic type list, the sub-type list, the sub-field list, the equivalence list, the recursive flag, and an auxiliary flag.

The action routines detailed below rely heavily on two subroutines; TypeEqual and IsRecursive. The exact algorithms for these routines are presented after the 4 action routines. For now, I will detail their format and effect.

TypeEqual has the format **TypeEqual (t1, t2)** where t1 and t2 are pointers to type nodes in the graph. TypeEqual returns yes if t1 and t2 point to equivalent nodes. Two types are equivalent if their basic type is equivalent, and all their sub-types are equivalent. TypeEqual returns no if t1 and t2 are inconsistent. Two types are inconsistent if their basic types incompatible OR at least one pair of their sub-types are incompatible. Note, the only

remaining possibility is that the two types are consistent. TypeEqual returns T1 if the information in union(t1,t2) is equal to the information in t1. TypeEqual returns T2 if the information in union(t1,t2) is equal to the information in t2. Otherwise, TypeEqual returns merge, meaning that the two types are consistent and each types contains information that the other does not. Note that TypeEqual cannot return merge unless both types are complex.

The format for IsRecursive is **IsRecursive (t)** where t is a pointer to a type node in the graph. IsRecursive returns 'no' if, by following all the sub-types to the end of their branches, the routine never returns to t again. Otherwise, it returns the recursive path or paths.

5.1.1 SetType

The format for SetType is **SetType (T, atomic _ type)**, where T is a pointer to a type node in the graph and atomic _ type is one of the atomic types.

step 1. If the basic type list of type # ~= {any} OR the atomic _ type is not an element of the basic type list THEN signal an error and return.

step 2. If the basic type = atomic _ type then return.

step 3. Set the basic type = Atomic _ type.

step 4. If the equivalence list for type # is empty OR the auxiliary flag is true, THEN return.

step 5. Set the auxiliary tag = true.

step 6. For each t_n in the equivalence list Do (a) call SetType (t_n,atomic _ type).

step 7. Set auxiliary tag = false and return.

The auxiliary flag is used by this routine to avoid following equivalence lists back and forth between type nodes. Note also that if T's basic type is {real,integer} and atomic _ type = integer, the basic type becomes {integer}.

Before the examples start, here is the format that these and all examples in this thesis will be in. There will be three sections for each example. The first will be labeled 'State', this section

contains the relevant information about the state of the data type graph. The second will be 'Action', this will be the action that is to occur. The last will be 'NewState', this will be the relevant information about the state of the data type graph after the action has occurred. Here are a few examples:

```
Note that the format for representing type nodes is:
Type number := {basic type list}  ( sub-type lists) (sub-field list)
                                   ={equivalence list}
If the sub-type, sub-field and equivalence lists are all empty,
they are omitted.

State:          T1 := {any}
Action:         SetType (T1,integer)
NewState:       T1 := {integer}

State:          T1 := {bool}
Action:         SetType (T1,bool)
NewState:       T1 := {bool}

State:          T1 := {real,integer}  ( () () )  ()  ={T2}
                T2 := {real,integer}
Action:         SetType (T1,real)
NewState:       T1 := {real} (()()) () ={T2}  ; T2 := {real}

State:          T1 := {char}
Action:         SetType (T1,empty)
NewState:       ERROR signalled

State:          T1 := {integer, char}
Action:         SetType {T1,real}
NewState:       ERROR signalled
```

5.1.2 SetMultType

The format for SetMultType is **SetMultType (T,typelist)** where T is a pointer to a type node in the graph and type list is a list of from 2 to 4 different types all of which are different and all of which are oneof (real, char, bool, integer).

Step 1. IF the basic type list of T is a subset of type list THEN return.

Step 2. IF the basic type list and typelist intersect, set the basic type list equal to the intersection and goto *Step 5*.

Step 3. IF the basic type ~ = {any} THEN signal an error and return.

Step 4. Set the basic type list = typelist.

Step 5. If the equivalence list is empty OR the auxiliary flag is true THEN return.

Step 6. Set the auxiliary tag = true.

Step 7. For each t_n in the equivalence list DO: Call SetMultType (t_n, typelist).

Step 8. Set auxiliary tag = false and return.

Similar to SetType, this routine uses the auxiliary flag to stop from infinitely traversing an equivalence list cycle. Here are a few examples of the effect of the routine.

```
State:          T1 := {any}
Action:         SetMultType (T1,{real,integer})
NewState:       T1 := {real,integer}

State:          T1 := {real}
Action:         SetMultType (T1,{real,integer})
NewState:       T1 := {real}

State:          T1 := {any}  (()()) () ={T2}
                T2 := {any}
Action:         SetMultType (T1,{integer,char})
NewState:       T1 := {integer,char}  (()()) () ={T2}
                T2 := {integer,char}

State:          T1 := {real,integer}
Action:         SetMultType (T1,{integer,char})
NewState:       T1 := {integer}

State:          T1 := {empty}
Action:         SetMultType (T1,{real,integer})
NewState:       ERROR signalled
```

5.1.3 Merge

Merge has the format **merge (T, complex_type)**, where T is a pointer to a type node in the graph, and complex_type is either a pointer to a complex type node in the graph or a pointer to a complex type node literal.

To make life easier, I am going to assume the call is merge (Ta, Tb) where Ta is T and Tb is the

complex_type.

Step 1. Call TypeEqual (Ta,Tb).

Step 2. If the result = 'yes' THEN insert Ta in Tb's equivalence list and Tb in Ta's equivalence list and return. (N. B. Do not make duplicate entries in equivalence lists.)

Step 3. If the result = 'no' THEN signal an error and return.

Step 4. If the result = 'T1' THEN set Tb's basic type = to Ta's basic type, set Tb's sub-type list = Ta's sub-type list, and set Tb's sub-field list = to Ta's sub-field list. Goto *Step 9.* (N. B. Step 9 detects recursive types.)

Step 5. If the result = 'T2' THEN set Ta's basic type = to Tb's basic type, set Ta's sub-type list = Tb's sub-type list, and set Ta's sub-field list = to Tb's sub-field list. Goto *Step 9.*

Step 6. (result = 'merge') (NB have to watch out for the special case of two multiple basic types needing merging) If Ta's basic type has more than one element, then set Ta and Tb's basic type := the intersection of their basic types and return. If Ta's basic type = array OR Ta's basic type = stream, then type_equal (Ta.sub-type,Tb.sub-type). (This is legal as arrays and streams can only have one sub-type in their sub-type list.)

Step 6b. If the result = 'yes' then return. If the result = 'no', then signal an error and return. If the result = 'T1', then set Tb.sub-type = Ta.sub-type and goto *Step 9.* If the result = 'T2', then set Ta.sub-type = Tb.sub-type and goto *Step 9.* If the result = merge, then call merge (Ta.sub-type, Tb.sub-type) and goto *Step 9.*

Step 7. If Ta's basic type = record OR oneof, then a) if a field is in Ta's sub-field list but not in Tb's, append the field to Tb's field list and append the corresponding sub-type to Tb's sub-type list; b) if a field is in Tb's sub-field list but not in Ta's, append the field to Ta's field list and append the corresponding sub-type to Ta's sub-type list; c) for all other fields (fields common to both types), call type_equal (Ta.sub-type, Tb.sub-type) on the corresponding sub-types. (N. B. All sub-types must be merged.)

Step 7a. If the result = 'yes' then loop. If the result = 'no' then signal an error and return. Else, call merge (Ta.sub-type, Tb.sub-type).

Step 7b. When *steps 7* and *7a* are completed, goto *Step 9*.

Step 8. (T1's basic type = function.) If a field is in Ta's sub-field list and the matching field is blank in Tb's, then place the field from Ta into the blank field's position in Tb. If a field is in Ta's sub-field list and not in Tb's then append the field onto Tb's list and append the corresponding sub-type onto the first of Tb's sub-type lists. (Please note, fields and types for function type nodes that are in the same position in the sub-field and sub-types lists are matching fields or matching types.) If a field is in Tb's sub-field list and the matching field is blank in Ta's, then place the field from Tb into the blank field's position in Ta. If a field is in Tb's sub-field and not in Ta's, then append the field onto Tb's list and append the corresponding sub-type onto the first of Ta's sub-type lists. (N. B. Remember that functions have two sub-type lists.)

Step 8a. Otherwise, for each pair of corresponding type in each pair of lists DO call type_equal (Ta.sub-type,Tb.sub-type). If the result = 'yes' then loop. If the result = 'no' then signal an error and return. Else, call merge (Ta.sub-type, Tb.sub-type).

Step 8b. When *Step 8b* is done, goto *Step 9*.

Step 9. Determine if Ta has become a recursive type. Call IsRecursive (Ta). If the result = 'no' then goto *Step 10*. (T1 is recursive.) If Ta's basic type = array OR stream OR function then signal an error and return. If Ta's basic type = oneof, then set the recursive flags = true for all the type nodes in the paths returned. Also, set Tb's recursive flag = true, then goto *Step 10*. (T1 is recursive and a record type.) Examine all type nodes in each path, if any path is all basic types = record OR any path is not all oneofs and records THEN signal an error and return. Otherwise, set recursive flags = true for all the type nodes in the paths returned, and for Tb. Goto *Step 10*. (N. B. Recursive types must contain union types to guard against infinite recursion.)

Step 10. Call equiv (Ta,Tb).

Step 11. If Ta's equivalence list is empty or the auxiliary tag = true or the list contains just the entry Tb, then goto *Step 12*. Set Ta's auxiliary tag = true. Otherwise, for all the t_n in the list, call merge (Ta,t_n).

Step 11a. Set Ta's tag = false. Goto *Step 12.*

Step 12. Do the analogous steps for Tb as in 11 and 11a.

Step 13. Return.

Here are a few examples of merge:

```
State:        t1 := {any} ;   t2 := {array} ( (Tc) () ) () ={}
              Tc := {any}
Action:       merge (t1,t2)
NewState:     t1 := {array} ( (Tc) () ) () ={t2}
              t2 := {array} ( (Tc) () ) () ={t1}

State:        t1 := {stream} ( (Tb) () ) () ={}
              t2 := {stream} ( (Tc) () ) () ={}
              Tb := {integer}
              Tc := {real}
Action:       merge (t1,t2)
NewState:     ERROR signalled

State:        t1 := {array} ( (Tb) () ) () ={}
              t2 := {array} ( (Tc) () ) () ={}
              Tb := {real}
              Tc := {real,integer}
Action:       merge (t1,t2)
NewState:     t1 := {array} ( (Tb) () ) () ={t2}
              t2 := {array} ( (Tb) () ) () ={t1}

State:        t1 := {record} ( (Ta Tb Tc) () ) ( 'a', 'b', 'c') ={}
              Ta := {integer}
              Tb := {any}
              Tc := {empty}
              t2 := {record} ( (Td Te Tf ) () ) ('b', 'c', 'd')  ={}
              Td := {char}
              Te := {stream} ( (Tg) () ) () ={}
              Tf := {null}
              Tg := {bool}
Action:       merge (t1,t2)
NewState:     t1 := {record} ((Ta Td Te Tf)()) ('a','b','c','d') ={t2}
              t2 := {record} ((Ta Td Te Tf)()) ('a','b','c','d') ={t1}

State:        t1 := {function} ((Ta Ta) (Tb) () ={}
              Ta := {integer}
              Tb := {any}
              t2 := {function} ((Tc Tc) (Td)) () ={}
              Tc := {any}
              Td := {real}
Action:       merge (t1,t2)
```

```
NewState:          t1 := {function} ((Ta Ta) (Td)) () ={t2}
                   t2 := {function} ((Ta Ta) (Td)) () ={t1}

State:             t1 := {record}  ((Ta Tb) ()) ('a', 'b') ={}
                   Ta := {integer}
                   Tb := {any}
                   t2 := {record}  ((t1 Tc) ()) ('b', 'c') ={}
                   Tc := {real}
Action:            merge (t1,t2)
NewState:          ERROR - 'Illegal Recursive Type' signalled
```

5.1.4 Equiv

The format for equiv is equiv (t1, t2) where t1 and t2 are both pointers to type nodes in the graph.

Step 1. Call TypeEqual (t1,t2).

Step 2. If the result = 'yes' THEN if t1's basic type = null OR real OR integer OR char OR bool then return. Else, put t1 in t2's equivalence list. Put t2 in t1's equivalence list and return.

Step 3. If the result = 'no' then signal an error and return.

Step 4. Put t1 and t2 into each others equivalence lists.

Step 5. If the result was = 'T1', set all of t2 equal to t1 except t2's equivalence list. If t2's equivalence list is empty, or t2's auxiliary flag = true, then return. Else, set t2's auxiliary flag = true. For all t_n in t2's equivalence list, equiv (t_n,t2). Return.

Step 6. If the result = 'T2', do as in *Step 5* except switch the t1's and t2's.

Step 7. The result was = 'merge'. Call merge (t1,t2). Return.

Note that the only times when equiv will not insert the type pairs into the equivalence lists is when the two types are not consistent or when they are equivalent and atomic types that are not any and not empty. (This is because type nodes that are any, or empty can be changed without an error.) Here are a few examples of equiv:

```
State:          t1 := {integer} ; t2 := {any}
Action:         equiv (t1,t2)
```

```
NewState:      t1 := {integer} (()()) () ={t2}
               t2 := {integer} (()()) () ={t1}

State:         t1 := {real} (()()) () ={t3, t7}
               t2 := {real}
Action:        equiv (t1,t2)
NewState:      no change

State:         t1 := {record} ((Ta Tb) ()) ('a','b') ={t3}
               Ta := {integer}
               Tb := {real, integer}
               t3 := {record} ((Ta Tb) ()) ('a','b') ={t1}
               t2 := {record} ((Tc Td Te) ()) ('a','b','c') ={}
               Tc := {any}
               Td := {real}
               Te := {empty}
Action:        equiv (t1,t2)
NewState:      t1 := {record} ((Ta Td Te) ()) ('a','b','c') ={t2, t3}
               t2 := {record} ((Ta Td Te) ()) ('a','b','c') ={t1}
               t3 := {record} ((Ta Td Te) ()) ('a','b','c') ={t1}

State:         t1 := {array} ((Ta)()) () ={}
               t2 := {stream} ((Ta)()) () ={}
               Ta := {integer}
Action:        equiv (t1,t2)
NewState:      ERROR signalled
```

As you may have noticed, the preceding four action routines call two subroutines; IsRecursive and TypeEqual. IsRecursive is responsible for determining if its argument is a recursive type or not. If it is, it passes back all the recursive paths that reach the type node of its argument. TypeEqual is responsible for determining how its two argument types are related. Are they exactly equivalent, consistent or inconsistent? If they are consistent, is one merely a subset of the other? The next sections present the exact algorithms for both of these routines.

5.1.5 IsRecursive

The format for this subroutine is **IsRecursive(t)**. Where t points to a complex type node in the graph. Note that this subroutine has an internal subroutine **findp** which is defined below it.

Step 1. If the basic type of t = array OR stream THEN pathlist = {t}. Call findp (pathlist, t.sub-type).

Step 2. If the basic type of t = record OR oneof THEN Cnt := 0, for all types in the sub-type list DO cnt := cnt + 1, pathlist.cnt := {t}, call findp (pathlist.cnt, t.sub-type).

Step 2a. If all results = false, return 'no'. Else return all pathlists returned by findp.

Step 3. (T's basic type = function.) Cnt := 0. For all types in both sub-type lists DO cnt := cnt + 1, pathlist.cnt := {t}, call findp (pathlist.cnt, t.sub-type).

Step 3a. If all results = false, return 'no'. Else return all pathlists returned by findp.

Here is the algorithm for findp. Note that it has the format findp (pathlist, t). (NB car(pathlist) returns the first element in the pathlist.)

Step 1. If t = car (pathlist) then return (pathlist)

Step 2. If t's basic type = any OR null OR char OR bool OR integer OR real OR empty return (false).

Step 3. Append t's type number to pathlist.

Step 4. If t's basic type = array OR stream THEN call findp (pathlist,t.sub-type).

Step 5. If t's basic type = record OR oneof THEN cnt := 0, for all types in t's sub-type list DO cnt := cnt + 1, pathlist.cnt := pathlist, call findp (pathlist.cnt, t.sub-type), loop.

Step 5a. If all the results are false, return (false). Else, return all the pathlists.

Step 6. (T has basic type = function). Cnt := 0, for all types in t's sub-type lists DO cnt := cnt + 1, pathlist.cnt := pathlist, call findp (pathlist.cnt, t.sub-type), loop.

Step 6a. If all the results are false, return (false). Else, return all the pathlists.

Here are a few examples of IsRecursive:

```
State:          t1 := {record} ((Ta)()) ('a') ={}
                Ta := {any}
Action:         IsRecursive (t1)
NewState:       'no' is returned

State:          t1 := {array} ((Ta)()) () ={}
```

```
                 Ta := {record}  ((Tb)()) ('b') ={}
                 Tb := {array} ((Tc) ()) () ={}
                 Tc := {any}
Action:          IsRecursive (t1)
NewState:        'no' is returned

State:           t1 := {stream} ((t2) ()) () ={}
                 t2 := {array} ((t1) ()) () ={}
Action:          IsRecursive (t1)
NewState:        (t1 t2) is returned ; N. B. is illegal recursive type

State:           t1 := {oneof} ((t2 Tb) ()) ('a', 'b') ={}
                 Tb := {record} ((Tc t3) ()) ('a', 'b') ={}
                 Tc := {null}
                 t2 := {record} ((t1) ()) ('b') ={}
                 t3 := {record} ((t2) ()) ('c') ={}
Action:          IsRecursive (t1)
NewState:        (( t1 t2 ) (t1 t3 t2)) is returned
                 (NB this is an example of a legal recursive type)
```

5.1.6 TypeEqual

The format for this subroutine is **TypeEqual (Ta, Tb)** where Ta and Tb are pointers to type nodes in the graph.

This subroutine has five possible results. A result of 'yes' means that the two types are equivalent. Two types are equivalent if and only if they have the same basic type, and all of their sub-types are equivalent. A result of 'no' means that the two types are not consistent. Two types are consistent (a) one of the types has the basic type 'any' OR (b) the two types are both atomic types and their basic types are equal OR (c) one or both of the types have two entries in their basic type lists AND the two basic type lists intersect OR (d) one of the types has basic type 'empty' and the other has basic type 'array' OR 'stream' OR (e) the basic types of both types are equal, AND sub-types and sub-fields common to both types are consistent. A result of 'T1' means that the types are consistent and that type Ta contains all the information found in type Tb. A result of 'T2' means that the types are consistent and that type Tb contains all the information found in type Ta. A result of 'merge' means that the types are consistent and that each type contains information not found in the other.

Step 1. If Ta has basic type any return ('T2'). If Tb has basic type any return ('T1'). (NB trivial cases first).

Step 2. If Ta has basic type null OR bool THEN if Ta's basic type = Tb's basic type return ('yes'). Otherwise, return ('no').

Step 3. If Ta has basic type real OR integer OR char THEN if Ta's basic type = Tb's basic type return ('yes'). If Ta's basic type is in Tb's basic type list, return ('T1'). Otherwise, return ('no').

Step 4. If Ta's basic type list has multiple elements THEN if Tb's basic type list = Ta's basic type list return ('yes'). If Tb's basic type list has multiple elements and it intersects with Ta, return ('merge'), else return ('no'). If Tb's basic type is an element of Ta's list, return ('T2'). Otherwise, return ('no').

Step 5. If Ta's basic type = empty THEN if Tb's basic type = empty return ('yes'). If Tb's basic type = array OR stream return ('Tb'). Otherwise, return ('no').

Step 6. If Ta's basic type = array OR stream THEN if Tb's basic type = empty, return ('Ta'). If Tb's basic type ~= Ta's basic type then return ('no'). Otherwise return (TypeEqual (Ta.sub-type,Tb.sub-type).

Step 7. If Ta's basic type = record OR oneof THEN if Tb's basic type ~= Ta's basic type then return ('no'). If Ta's recursive flag AND Tb's recursive flag are true, goto *Step 9* to handle recursive type problem.

Step 7a. For all fields in both Ta and Tb's field list DO if TypeEqual (Ta.sub-type, Tb.sub-type) = no, return ('no'). If all ~= yes, then return ('merge'). If arity Ta's field list ~= arity Tb's field list, then return ('merge'). Otherwise, return ('yes').

Step 8. (T1's basic type = function). If Tb's basic type ~= function return ('no'). If Ta's in sub-type list arity ~= Tb's in sub-type list arity then return ('no'). If Ta's out sub-type list arity ~= Tb's out sub-type list arity AND Ta's out arity ~= 0 AND Tb's out arity ~= 0 THEN return ('no'). Tresult = yes.

Step 8a. For all fields in Ta's field list DO if Ta.field = Tb.field list then loop. If Ta.field is blank or Tb.field is blank then tresult = merge and loop. Otherwise return ('no').

Step 8b. For all types in both of Ta's sub-type lists DO call TypeEqual (Ta.sub-type,Tb.sub--

type). If result = yes then loop. If result = no then return 'no'. Otherwise, tresult = merge.

Step 8c. If Ta's out sub-type list arity = Tb's out sub-type list arity, then return tresult. Otherwise, return merge.

Step 9. This is the algorithm to test consistency of two recursive types.

substep 1. Call IsRecursive (Ta). Save the returned pathlist. Remember that pathlist is the list of all recursive paths.

substep 2. Call IsRecursive (Tb). Save the returned pathlist.

substep 3. Construct a list of all possible pairs of nodes such that the first is from the Ta path and the second if from the Tb path. The first pair in the list should be (Ta Tb). Call this list the master list.

substep 4. Turn on Ta and Tb's auxiliary flag. Delete the first pair from the master list.

substep 5. Perform TypeEqual algorithm precisely as above, except ignore recursive flags for types in the pathlists and if TypeEqual should be called with one or more of its arguments with their auxiliary flags on then goto next substep.

substep 6. Here, TypeEqual has been called and one or more of the arguments has their auxiliary flags on. If Ta and Tb are both flagged, then turn their flags off, and return ('yes').

substep 7. Otherwise, check to see if Ta,Tb pair is in the master list. If not, turn off flag and return ('yes'). Otherwise, turn on Ta, Tb flag. Delete Ta, Tb pair from master list. Call TypeEqual (Ta, Tb) as in *substep 5.*

EXPLANATION - The requirement for consistency of two types is that the two type nodes must be consistent and that all sub-type nodes must also be consistent. With two recursive types, tracing consistency of sub-types causes the normal algorithm to loop around the recursive chain infinitely. To avoid this, the algorithm uses the auxiliary flag to mark where it started checking for consistency. If both recursive types return to their beginnings at the same time, then clearly they are consistent. However, if they return at different times, it does not follow that they must be inconsistent. What the algorithm must do is keep checking to see if the

sub-types are consistent. To do this, it resets the auxiliary flags and begins checking anew. The danger with this is that the algorithm may then reach the original start point and not realize that it has already checked this pair of type nodes for consistency. The master list is used to keep track of what pairs in the recursive paths have not been checked. In the worst case, all possible pairs in the paths will be checked. However, since the number of nodes in a data flow graph is finite, the number of distinct nodes in a recursive path is finite and the number of distinct recursive paths are finite. It follows that the number of possible pairs of nodes in recursive paths is finite. This means that the master list is finite and that the algorithm will close. This algorithm may seem to be somewhat inefficient. However, I know of no other algorithm that is any better.

There is one saving grace of the recursive algorithm. If one searches the graph in a bottom-up breadth first manner, the only places that recursive types will be created are at the last few nodes examined. This is because recursive types can only be created by the algorithm when the type's structure is searched via recursive calls on a function. A recursive call on a function is only detected when the operator node for the function header, the *Proced* node, is examined and its constraints applied. Proced nodes are always at the top of the graph, therefore recursive types are only formed at the top of the graph.

Here are a few examples of TypeEqual:

```
State:          t1 := {integer}
                t2 := {integer}
Action:         TypeEqual (t1, t2)
NewState:       'yes' is returned

State:          t1 := {null}
                t2 := {any}
Action:         TypeEqual (t1, t2)
NewState:       'T1' is returned

State:          t1 := {stream} ((Ta) ()) () ={}
                Ta := {any}
                t2 := {stream} ((Tb) ()) () ={}
                Tb := {real}
Action:         TypeEqual (t1,t2)
NewState:       'T2' is returned

State:          t1 := {oneof} ((Ta Tb) ()) ('a', 'b') ={}
                Ta := {char}
                Tb := {any}
                t2 := {oneof} ((Tc Td) ()) ('a', 'b') ={}
```

```
                 Tc := {real, integer}
                 Td := {array} ((Tb) ()) () ={}
Action:          TypeEqual (t1, t2)
NewState:        'no' is signalled

State:           t1 := {record}  ((Ta Tb) ()) ('a', 'b') ={}
                 Ta := {bool}
                 Tb := {record}  ((Tc) ()) ('q') ={}
                 Tc := {any}
                 t2 := {record}  ((Tc Td) ()) ('a', 'b') ={}
                 Td := {record}  ((Te) ()) ('q') ={}
                 Te := {integer}
Action:          TypeEqual (t1,t2)
NewState:        'merge' is returned
```

Here are a few examples with recursive types :

```
State:           t1 := {oneof} ((Ta t3) ()) ('a', 'b') ={}
                 Ta := {null}
                 t3 := {record} ((t1 Tb) ()) ('rest', 'val') ={}
                 Tb := {integer}
                 t2 := {oneof} ((Ta Tc) ()) ('a', 'b') ={}
                 Tc := {record} ((t2 Tb) ()) ('rest', 'val') ={}
Action:          TypeEqual (t1, t2)
NewState;        'yes' is returned

State:           t1 := {oneof} ((Ta t1) ()) ('a', 'b') ={}
                 Ta := {integer}
                 t2 := {oneof} ((Ta t3) ()) ('a', 'b') ={}
                 t3 := {oneof} ((Tb t2) ()) ('a', 'b') ={}
                 Tb := {real}
Action:          TypeEqual (t1, t2)
NewState:        'no' is returned

State:           t1 := {oneof} ((t3 t4 Ta) ()) ('a', 'b', 'c') ={}
                 Ta := {any}
                 t3 := {record} ((t1 Ta) ()) ('next', 'val') ={}
                 t4 := {record} ((t1 Tb) ()) ('next', 'val') ={}
                 Tb := {real}
                 t2 := {oneof}  ((t4 t3 Ta) ()) ('a', 'b', 'c') ={}
Action:          TypeEqual (t1, t2)
NewState:        'merge' is returned
```

6 Examples

This section presents three examples of the algorithm in action. The first example shows the old version of a VAL program, the new version and its data flow graph representation. It then shows how the operator nodes are examined and what their action routine calls would be. For each set of action routine calls, example 1 shows what new type information is gained. The second example focuses on the last action routine call from the first example. The call is followed in detail through all the steps of the precise algorithm. The third example is similar to the first in format, but in this case, an error is detected.

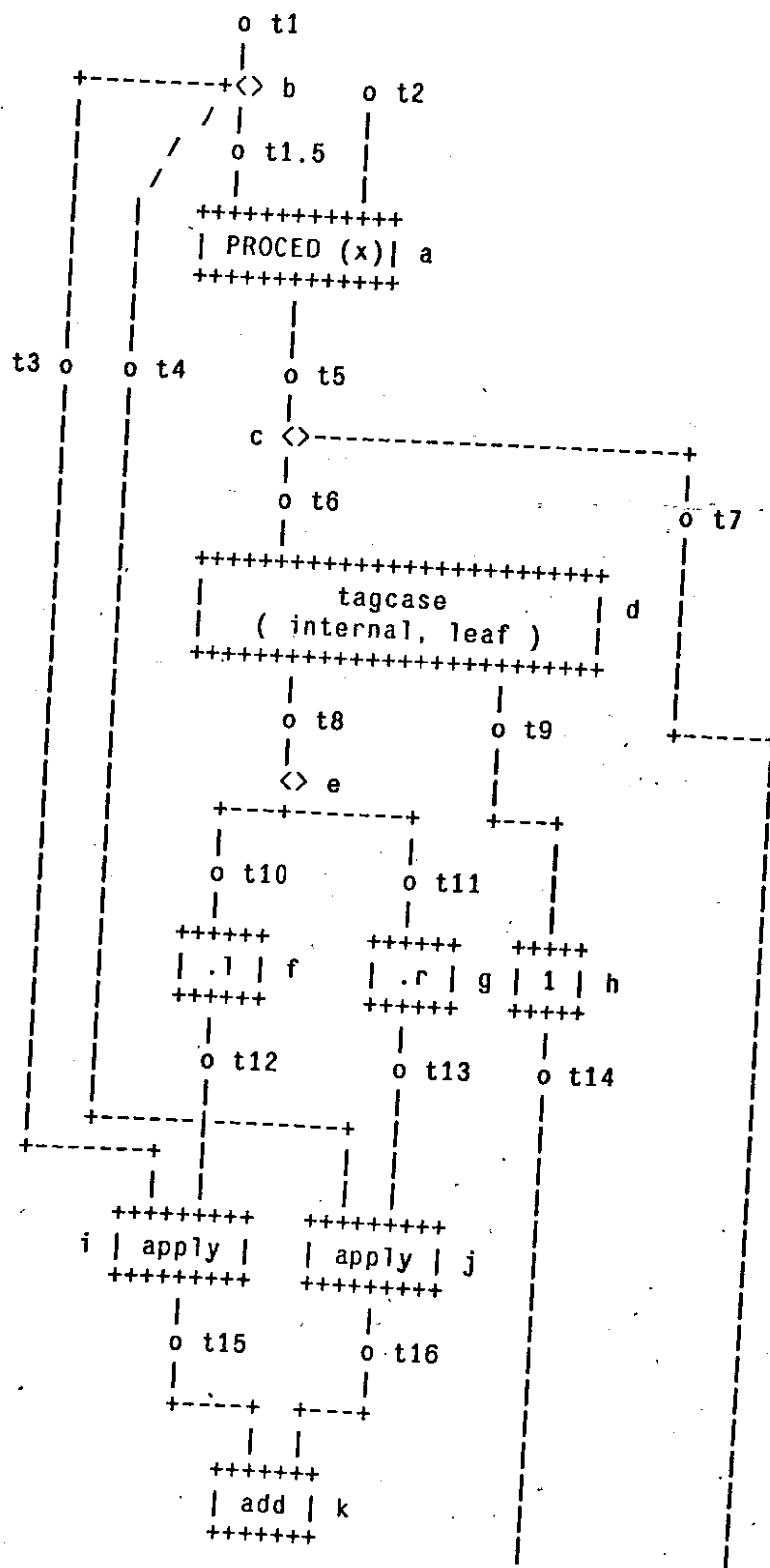
6.1 Example 1

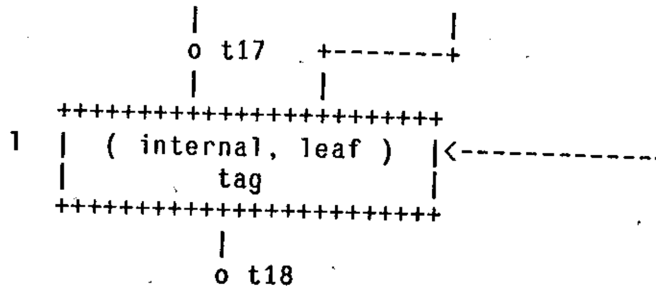
```
function CountLeaves ( x : tree returns integer)
type tree = oneof [ leaf : null ; internal : record
                          [l : tree; r : tree]];
tagcase y := x;
        tag leaf : 1
        tag internal : CountLeaves (y.l) + CountLeaves (y.r)
        endtag
endfun
```

N. B. This function counts the # of leaves on a tree structure.

```
function CountLeaves (x)
tagcase y := x;
        tag leaf : 1
        tag internal : CountLeaves (y.l) + CountLeaves (y.r)
        endtag
endfun
```

Here is the data flow graph for CountLeaves.





Note that all o's are type nodes, and that all have been given distinct type numbers. All operator nodes are boxes, and have been given identifying letters. These letters are for the purpose of these examples only. The symbol table would have three entries in it; CountLeaves, which would point to t1; x, which would point to t2; and y, which would also point to t2. All of the type nodes would be initialized to basic type any. Also note that the operator node <> is a link node which serves to divide one node into many.

Now the algorithm begins its bottom-up breadth first search of the data flow graph. The first operator node it hits is node l, a *tag* node.

A tag node is one of two nodes used to represent a tagcase construct. Its purpose is to join together the several cases of a tagcase and return the appropriate value based on the union type parameter. Its action routines calls are:

```
merge (t7, t0) where t0 is a pointer to the literal :
oneof [internal : any; leaf : any]
equiv (t17,t14)
equiv (t17,t18)

The resulting type information is :
t7 := oneof [internal : any; leaf : any]
t14 := any  = {t17}  ; where = { ... } is the equivalence list
t17 := any  = {t14, t18}
t18 := any  = {t17}
```

The next operator node examined is k, an *add* node.

```
Its action routine calls are:
SetMultType (t15,{real, integer})
equiv (t15, t16)
equiv (t15, t17)

The new type information is: (only if info for that type has changed)

t14 := (real, integer)  = {t17}
```

```
t15 := (real, integer)  = {t16, t17}
t16 := (real, integer)  = {t15}
t17 := (real, integer)  = {t14, t15, t18}
t18 := (real, integer)  = {t17}
```

The next operator node expanded is i, an *apply* node. An apply node is used to handle calls on functions.

Its action routine calls are :
merge (t3, t0) where t0 is the literal :
t0 := function ((t12) (t15)) (' ')
(Meaning t0 is a function taking t12 as an argument, with an unknown identifier, and returning t15.)

The new type information is :
t3 := function ((t12) (t15)) (' ')

The next operator node examined is j, another apply node.

Its action routine call is :
merge (t4, t0) where t0 is the literal
t0 := function ((t13) (t16)) (' ')

The new type information is :
t4 := function ((t13) (t16)) (' ')

The next operator node examined is f, a *record select* node.
Its action routine call is :
merge (t10, t0)
t0 := record [1 : t12]

The new type information is :
t10 := record [1 : t12]

The next operator node examined is g, another record select node.

Its action routine call is :
merge (t11, t0)
t0 := record [r : t13]

The new type information is :
t11 := record [r : t13]

The next operator node examined is h, an integer constant node.

Its action routine call is :
SetType (t14, integer)

The new type information is :

```
t14 := integer  = {t17}
t15 := integer  = {t16, t17}
t16 := integer  = {t15}
```

```
t17 := integer  = {t14, t15, t18}
t18 := integer  = {t17}
```

The next operator node examined is e, a *link* node.

Its action routine calls are :
equiv (t8, t10)
equiv (t8, t11)

The new type information is :
```
t8 := record [ l : t12; r : t13 ]  = {t10, t11}
t10 := record [l : t12; r : t13 ]  = {t8}
t11 := record [l : t12; r : t13 ]  = {t8}
```

The next operator node examined is d, a *tagcase* node. The tagcase node is used to break up the original union type into its constituent parts, as is done by the tagcase construct.

Its action routine call is :
merge (t6, t0)
t0 := oneof [internal : t8, leaf : t9]

The new type information is :
t6 := oneof [internal : t8, leaf : t9]

The next node examined is c, another link node.

Its action routine calls are :
equiv (t5,t6)
equiv (t5,t7)

The new type information is : N.B. t6 and t7 are consistent
```
t5 := oneof [internal : t8, leaf : t9 ]  = {t6, t7}
t6 := oneof [internal : t8, leaf : t9 ]  = {t5}
t7 := oneof [internal : t8, leaf : t9 ]  = {t5}
```

The next node examined is b, another link node.

Its action routine calls are :
equiv (t1, t3)
equiv (t1, t4)
equiv (t1, t1.5)

The new type information is : (t3, and t4 are equivalent as t12 = t13 = any and t15 = t16 = integer.)
```
t1 := function ( (t12) (t15) ) ( ' ' )  = {t1.5, t3, t4}
t1.5 := function ( (t12) (t15) ) ( ' ' )  = {t1}
t3 := function ( (t12) (t15) ) ( ' ' )  = {t1}
t4 := function ( (t13) (t15) ) ( ' ' )  = {t1}
```

The last type node examined is a, a *proced* node. A proced node is used to represent a function header.

```
Its action routine calls are :
equiv (t2, t5)
merge (t1.5,t0)
t0 := function ( (t2) () ) ( 'x' )

The resulting type information is :
t1 := function ( (t12) (t15) ) ( 'x' )  = {t1.5, t3, t4}
t1.5, t3 and t4 are similar.
t2 := oneof [internal : t8; leaf : t9 ]  = {t5}
t8 := record [ l : t12 ; r : t13 ] = {t10, t11}
t9 := any
t12, t13 := oneof [internal : t8; leaf : t9]  = {each other}
t15 := integer = { t16, t17}
```

Note that not all of this information is new, it was all included for the sake of clarity. Note also that t2, t8, t12, and t13 point to recursive types. To get an understanding of how the merge of t1.5 and t0 resulted in t12, t13 being changed, see example 2.

Because the algorithm reached this point, the module is considered to be type checked. However, that does not mean that the algorithm is type determined. In fact, if one follows through all the types in the symbol table to make sure that they are all type determined, one quickly finds out that the oneof type is undetermined as the field 'leaf' is pointing to an 'any' type. Type determination for this module will have to wait to link time.

6.2 Example 2

The action routine call is merge (t1.5, t0).

```
t1.5 := function ( (t12) (t15) ) ( ' ')  = {t1}
t12 := any
t15 := integer  = {t16, t17}
t1 := function ( (t12) (t15) ) ( ' ' ) = {t1.5, t3, t4}
t3 := function ( (t12) (t15) ) ( ' ' )  = {t1}
t4 := function ( (t13) (t15) ) ( ' ' )  = {t1}
t13 := any
t16, and t17 are irrelevant

t0 := function ( (t2) () ) ( 'x')
t2 := oneof [internal : t8; leaf : t9]  = {t5}
t8 := record [l : t12; r : t13]  = {t10, t11}
t9 := any
t5, t10, and t11 are irrelevant
```

Entry MERGE1 Ta bound to t1.5, Tb bound to t0

ME1.1) Call TypeEqual (Ta, Tb)

Entry TE1 Ta bound to t1.5, Tb bound to t0

TE1.1) Neither Ta or Tb has basic type any.

TE1.2) Ta is not basic type null or bool.

TE1.3) Ta is not basic type real or integer or char.

TE1.4) Ta's basic type list has only one element.

TE1.5) Ta is not basic type empty.

TE1.6) Ta is not basic type array or stream.

TE1.7) Ta is not basic type record or oneof.

TE1.8) Tb's basic type is equal to record. Ta's input sub-type list arity does equal Tb's input sub-type list arity. Tb's out sub-type list arity does equal 0. TE1.tresult = yes.

TE1.8a) ' ' ~= 'x'. ' ' is a blank. Tresult = merge.

TE1.8b) Call TypeEqual (t12, t2)

Entry to TE2, Ta is bound to t12, Tb is bound to t2.

TE2.1) Ta has basic type any. Returning 'yes'.

Exit from TE2, 'Tb' is returned.

(still in TE1.8b) TE1.tresult = merge.

TE1.8c) Ta's out sub-type list arity ~= Tb's out sub-type list arity. Returning 'merge'.

Exit from TE1, 'merge' is returned.

(now back in original merge call)

ME1.2) Result ~= 'yes'.

ME1.3) Result ~= 'no'.

ME1.4) Result ~= 'Ta'.

ME1.5) Result ~= 'Tb'.

ME1.6) Ta's basic type list does not have multiple elements and is not equal to array or stream.

ME1.7) Ta's basic type does not equal record or oneof.

ME1.8) Place 'x' in the blank field in Ta. Note that t1.5 := function ((t12) (t15)) ('x') now.

ME1.8a) Call TypeEqual (t12, t2).

Entry point to TE1, Ta bound to t12, Tb bound to t2.

TE1.1) Ta has basic type = 'any', return a 'yes'.

Exit point from TE1, returning a 'yes'.

(still in ME1.8a) Call merge (t12, t2)

Entry point to ME2, where Ta = t12, and Tb = t2.

ME2.1) Call TypeEqual (t12, t2).

Entry point to TE1, where Ta = T12, Tb = t2.

TE1.1) Ta has basic type 'any', return 'Tb'.

Exit from TE1, result = 'Tb'.

ME2.2) Result does not equal 'yes'.

ME2.3) Result does not equal 'no'.

ME2.4) Result does not equal 'Ta'.

ME2.5) Result does equal 'Tb'. Result of step 5 is that t12 := oneof [internal : t8; leaf : t9] now.

ME2.9) Call IsRecursive (t12).

Entry to IR1, T bound to t12.

IR1.1) T12's basic type does not equal array or stream.

IR1.2) T12's basic type does equal oneof. Call findp (pathlist.1,t8) where pathlist.1 = {t12, t8}.

Entry to FI1, pathlist is bound to {t12}, t is bound to t8.

FI1.1) t8 does not equal t12.

FI1.1.5) t is not a member of pathlist.

FI1.2) t is a complex type.

FI1.3) pathlist := {t12, t8}

FI1.4) T's basic type is ~= to array or stream.

FI1.5) t's basic type does equal record. Call findp (pathlist.1, t12), where pathlist.1 = {t12, t8}.

Entry to FI2, pathlist bound to {t12, t8}
T bound to t12.

FI2.1) t12 does equal t12. Return pathlist.

Exit from FI2, result = {t12, t8}

(still in FI1.5). Result = {t12, t8}. Call findp (pathlist.2,t13), where pathlist.2 = {t12, t8}

Entry to FI2, pathlist bound to {t12, t8}
T bound to t13.

FI2.1) t13 does not equal t12.

FI2.2) t13 is not a member of {t12, t8}

FI2.3) t13's basic type is equal to 'any'. Return false.

Exit from FI2, result = 'false'.

(still in FI1.5). Return result.

Exit from FI1. Result = {t12, t8)

(still in IR1.2). Result = {t12, t8}. Call findp (pathlist.2, t9). Where pathlist.2 = {t12}.

Entry to FI1, pathlist bound to {t12}, T bound to t9.

FI1.1) t9 does not equal t12.

FI1.2) t9 is not a member of {t12}.

FI1.3) t9's basic type = 'any'. Return false.

Exit from FI1, result = 'false'.

(still in IR1.2). return result.

Exit from IR1, result = {t12, t8}

(still in ME2.9). Result ~= 'no'. Ta's basic type does not equal array, or stream or function. Ta's basic type is equal to oneof. Set t12, t2 and t8's recursive flags to true.

ME2.10) Call equiv (t12, t2).

Entry to EQ1, Ta bound to t12, Tb bound to t2.

EQ1.1) Call TypeEqual (t12,t2)

TypeEqual call omitted. Result = 'yes'.

EQ1.2) Put t12 in t2's each others equiv lists
t12 := = {t2}, t2 := = {t5, t12}.

Exit from EQ1.

ME2.11) t12's equivalence list contains just t2.

ME2.12) t2's equiv list contains t5. Set t2's aux tag = true.
Call merge (t2, t5). Set t2's aux tag to false.

Merge call omitted, as has no effect on types.

(Still in ME 2.12). Return.

Exit from ME2.

(still in ME1.8a). The corresponding type to Ta.t15 in Tb is null.
Copy t15 into the list. Note that t0 := ((t2) (t15)) ('x') now.

ME1.9) Call IsRecursive (T1.5).

IsRecursive call omitted, result = 'no'.

(Still in ME1.9). Result is equal to 'no'.

ME1.10) Call Equiv (t1.5, t0)

Equiv call omitted as irrelevant.

ME1.11) Call merge (t1.5, t1).

Note this is the last step from the example that is explicitly shown. However, what occurs inside this merge is a call on merge (t1, t4). This merge is what provides the information that t12, and t13 are the same type, and also that t13 is a recursive type. If the reader wants to assure that this is true, just follow the detailed steps of the algorithm.

6.3 Example 3

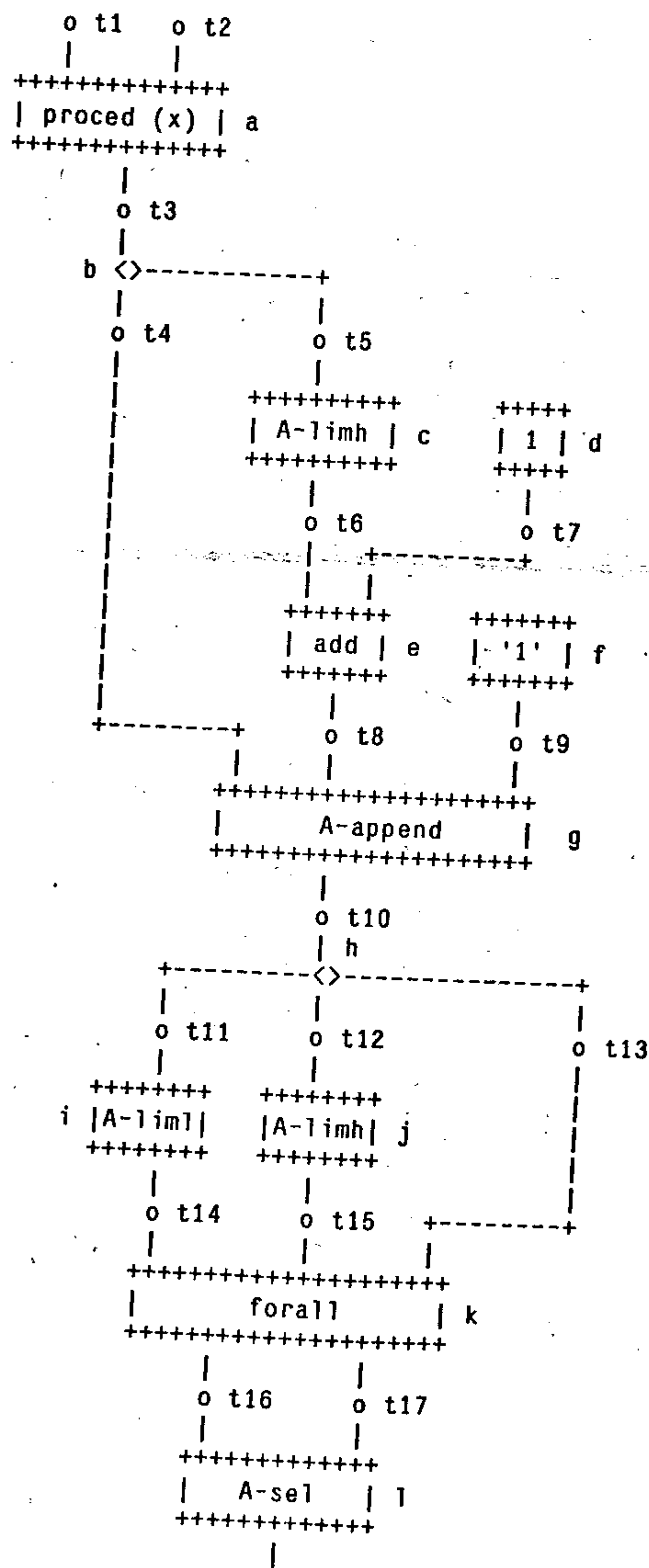
This example is a program with a built in type error. Specifically, in the function badtype below, the array 'a' is being used both as an array of integers and an array of chars. Incidentally, what the function is trying to return is the 1 + the sum of the elements of an array. Here is the function including the type declarations :

```
function badtype ( x : array [integer] returns integer )
        let a : array [integer];
           a := x [ (array_limh (x) + 1) : '1'] ;
        in forall i in [array_liml(a),array_limh(a)]
           eval plus a[1]
           endall
        endlet
endfun
```

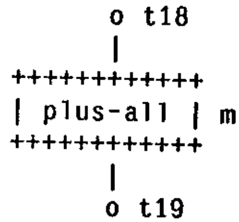
Here is the function without the type declarations :

```
function badtype ( x )
        let a := x [ (array_limh (x) + 1) : '1'] ;
        in forall i in [array_liml(a),array_limh(a)]
           eval plus a[1]
           endall
        endlet
endfun
```

Here is the data flow graph which represents the program.



o t1
o t2
proced (x)
a
o t3
b
o t4
o t5
A-limh
c
1
d
o t6
o t7
add
e
'1'
f
o t8
o t9
A-append
g
o t10
h
o t11
o t12
o t13
i
A-liml
A-limh
j
o t14
o t15
forall
k
o t16
o t17
A-sel
l



The symbol table has two entries. The entry badtype points to t1. The entry x points to t2. Here is the way the general algorithm finds the error.

The first operator node examined is m, a *plus-all* node. Plus-all nodes are one of a class of nodes that perform the eval function inside a forall construct.

Its action routine calls are:

```
SetMultType (t18, {real, integer})
equiv (t18, t19)
```

The new type information is:

```
t18 := {real, integer}  ={t19}
t19 := {real, integer}  ={t18}
```

The next operator node examined is l, an *A-sel* node. A-sel nodes are array select operations.

Its action routine calls are:

```
SetType (t16, integer)
merge (t17, t0) ; where t0 := array [t18]
```

The new type information is:

```
t16 := integer
t17 := array (t18)
```

The next operator node examined is k, a *forall* node. Forall nodes are used to represent the forall header. They also funnel the limits to a single integer type.

Its action routine calls are:

```
SetType (t14, integer)
SetType (t15, integer)
SetType (t16, integer)
equiv (t13, t17)
```

The new type information is:

```
t13 := array (t18)  ={t17}
t14 := integer
t15 := integer
t17 := array (t18)  ={t13}
```

The next operator node examined is i, an *array-liml* node. Array-liml nodes return the low index of their argument arrays.

Its action routine calls are:
SetType (t14, integer)
merge (t11, t0) ; where t0 := array [any]

The new type information is:
t11 := array [any]

The next operator node examined is j, an *array-limh* node. Array-limh nodes return the high index of their argument arrays.

Its action routine calls are:
SetType (t15, integer)
merge (t12, t0) ; where t0 := array [any]

The new type information is:
t12 := array [any]

The next operator node examined is h, a link node.

Its action routine calls are:
equiv (t10,t11)
equiv (t10,t12)
equiv (t10,t13)

The new type information is :
t10 := array (t18) ={t11, t12, t13}
t11 := array (t18) ={t10}
t12 := array (t18) ={t10}
t13 := array (t18) ={t10, t17}

The next operator node examined is g, an *A-append* node. This node performs the array append operation.

Its action routine calls are:
SetType (t8, integer)
merge (t4, t0) ; where t0 := array [t9]
equiv (t4, t10)

The new type information is:
t4 := array (t9) ={t10}
t8 := integer
t9 := {real, integer} ={t18}
t10 := array (t18) ={t4,t11,t12,t13}

The next operator node examined is e, an add node.

Its action routine calls are:
SetMultType (t6, {real, integer})

```
equiv (t6, t7)
equiv (t6, t8)

The new type information is:
t6 := integer  ={t7}
t7 := integer  ={t6}

The next operator node examined is f, a char constant.

Its action routine call is:
SetType (t9, char)

SetType signals an error.
```

At this point the type-checker can do one of two things. It can blow up after giving an appropriate error message, or it an assume one assignment or the other is correct and print the error message and keep on going. Provided that good assumptions can be made, the second course is far and away the better, as it allows the programmer to see all of the errors in a module with one compilation attempt.

7 Module Link Algorithm

So far, no algorithm has been specified that will allow modules that are not type determined to be further type checked at module link time. Module link time is when two or more VAL modules are loaded together. Any identifier in one module that refers to a value in another module, must be *bound* to that value. For example, an identifier 'foo', that refers to a function 'foo', in a module 'bar' would have to be bound to the function definition. The concern for the type checker is to insure that such a binding would be type consistent, and that, after all such bindings are finished, the modules are type determined.

7.1 Type information available

Before discussing what exactly a bind will do, it is useful to review what type information will be available at link time. A module, regardless of whether or not it is type determined, has all its type information available through its symbol table. The symbol table contains an entry for each formal parameter and free variable in the module. Each entry contains an identifier and a pointer to a type node that corresponds to that identifier. As before, the type node has

associated with it a type specification and an equivalence list. All type nodes that are referenced directly by the symbol table and indirectly through type specifications are available. The information formerly present in the data type graph, but not available at link time, is the set of operator nodes and all the type nodes which are not referenced. This information is not needed in order to bind data types.

7.2 Binding

Currently, binding is envisioned as occurring from specific, programmer generated calls on a function *bind*. The function 'bind' would take as its arguments the name of the non-determined module and a list of pairs, the first of which is a non-determined type in the module, and the second of which is the module to which the type should be bound. Note, it is a requirement of VAL that the second module be type determined. The function 'bind' is defined as doing:

1) Testing to see if the non-determined type and the type to which it is to be bound are type consistent. The type to which the non-determined type is to be bound, the 'binding' type, is found by looking at the symbol table for the specified module, the binding type will have the same identifier as the non-binding type. If the binding type is not found, or it is not type consistent with the non-determined type, an error should be signalled.

2) Perform a merge on the type nodes pointed to by the non-determined type and the binding type.

3) Test to see if both types are type determined now. If not, a message to that effect should be signalled.

Note that step 3 is necessary as the test in step 1 is consistency, so the non-determined type may contain sub-types not in the binding one. This would lead to the types being non-determined, even after the merge is performed.

7.2.1 Assertion

Using the bind function defined above, with the information described above, it is possible to bind modules which are not type determined to functions which are type determined in such a

way as to cause the first module to be type determined. Also, that if such a binding occurs with no error, the type specifications created will not cause corresponding types (types with the same identifiers) to be type inconsistent.

1) The second module is type determined, thus all its types are type determined.

2) If the non-determined type is consistent with the determined one, and all of the non-determined type's are in the determined one's specification or are referenced by that specification, after the merge both type's will be determined.

3) When two type nodes are merged, the resulting type specifications are consistent.

Step 2 implies that type determinacy can be reached. Step 3 says that type consistency is preserved.

7.3 Example

Here is an example of the bind function:

```
The symbol table for a module 'foo' has 4 entries, 'foo', 'x', 'y',
and 'add'.  Foo, x and y are type determined.  Add is not.

add -> {function}  ( (t4 t5) (t6)) () ={}
t4 -> {real, integer}
t5 -> {real, integer}
t6 -> {real, integer}

There is a module 'adder' that had the identifier 'add' in its
symbol table. Adder is type determined.  The type information
for add is:

add -> {function} ((t1 t2) (t3)) ()
t1, t2, t3 -> integer

The programmer enters the command:
bind (foo, ( (add adder)))

The result is that the two adds are merged and that
t4,t5,t6 -> integer.

Foo is now type determined.
```

8 Producing Data Type Graphs

It is not the intention of this thesis to present an algorithm for translating VAL modules from their text form into data type graphs. As has been stated before, data type graphs derive from data flow graphs. The translation of textual programs into data flow graphs has already been adequately handled in [Weng 75]. What will be presented here is a brief synopsis of how Weng proposes to do translation and then a short discussion of how to extend Weng's method in order to produce data type graphs.

8.1 Data Flow Graphs

Weng's translation rules depend on the ability to recognize the input and output of a statement. The input are those values whose identifiers are referenced but not defined in the statement. The outputs are those values that are defined in the statement.

If L is a list of statements (one way of looking at a program module), then the inputs of L are all identifiers referenced but not defined in L, and the outputs of L are elements in the union of the outputs of each of the statements in L.

For an expression that contains no constructs (examples of constructs are, IF THEN ELSE, TAGCASE, FORALL, etc.), translation to graph form is simple. An acyclic graph is created according to the rules of evaluation of the operators which make up the expression. The inputs of the operators are connected to the links labeled with the identifiers appearing in the statement. The outputs are connected to wherever the result of the expression goes (an assignment, or maybe a returned value, or whatever),

Construct translation, while trickier is also straightforward. Each construct usually requires the creation of one or two special operator nodes. In the case of the IF THEN ELSE, the nodes required are T, F, and TF nodes. The detailed rules for translation are not presented here. The reader is invited to read Weng's paper or examine the Appendix for examples of translation. Suffice it to say that construct translation is mostly just expression translation and rules for tying together the inputs and outputs of the expressions.

8.2 Data Type Graphs

Data flow graphs are not data type graphs. The data flow graphs contain no type nodes, just links labeled with identifiers. To create a data flow graph from a data type graph is simple. Merely insert a type node between each pair of linked operator nodes in the graph. If a labeled link is representing a formal parameter or free variable, place the identifier in the symbol table and point the table entry at the argument type node to the link operator. Remove all link labels.

Each type node is initialized with an empty equivalence list and a type specification with a basic type list equal to {any}. If an identifier had a type declaration, that information is incorporated into the specification in place of the default, 'any'.

To sum up, Weng's work shows how to translate text programs into data flow graphs. The translation from data flow graphs to data type graphs is trivial. Thus, it follows, that producing data type graphs from VAL text is both possible and straightforward.

9 Conclusion

The programming language VAL is currently undergoing revision. Some of the proposed changes have a severe impact on the ability of the VAL compiler to do all type checking at compile time. The most notable of these changes are the elimination of required type declarations, the inclusion of function data types, and the provision that recursive data types must contain union types.

This thesis has presented one solution to the problem of how to type check the new version of VAL. The basis for solution is the analysis of data type graphs. These graphs are just data flow graphs with the inclusion of type nodes. The graphs are analyzed by examining operator nodes to see what type constraints the nodes place on their argument and result type nodes. By properly applying and preserving these constraints, enough type information may be deduced so as to allow the module to be type checked at compile time. Furthermore, all the type information determined allows the modules to be correctly and uniquely type determined at module link time.

9.1 Alternatives

Clearly, the approach of this thesis is not the only one. Another approach might be to examine the VAL module without translating it into an alternate representation. VAL modules can be type checked straight from their text form.

However, I feel that the translation into graph form is a powerful tool for simplifying the type checker's task. The algorithm is easy to develop as it only has to concern itself with two types of nodes. The graph makes constraint propagation (via equivalence lists) easier to visualize. The tradeoff in graph production considerations is not bad. VAL was originally designed for a data flow architecture, so the translation turns out to be simple.

9.2 Applications to Other Languages

This thesis has been concerned with type checking generalized VAL, yet the algorithm developed herein can be applied to other languages. The essence of the algorithm is that the types of data values can be determined by examining the operations that are to be performed on them. Furthermore, most languages can be translated into data type graphs of a form similar to the one presented in this thesis.

The key to the algorithm was the type constraints produced by the operators. I believe that similar constraints can be easily developed in most languages. Certainly CLU, PL/1 and other strongly typed languages seem amenable to this sort of approach. I also believe, though with less certainty, that LISP, APL and other such languages can also be type checked in this manner.

9.3 Further Work

There are two clear areas where further work needs to be done on this algorithm. One, the algorithm has not been formally proved. This is essential if it is to be trusted to always work correctly. Two, the algorithm needs to be implemented. This, too, is obvious as one can never be sure that an algorithm can be implemented until it has.

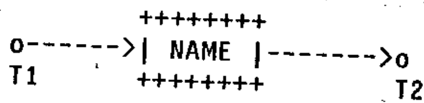
Appendix A

VAL Operators and Constructs

10 Basic Operators

10.1 Error Tests

There are three universal error tests in VAL. Their names are *is undef*, *is miss-elt*, and *is error*. Their graph representation is:



The only type constraint is that T2 must be type 'bool'.

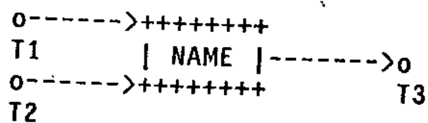
The action routine call would be:

```
SetType (T2,bool)
```

10.2 Equal and Not Equal

Equal, ' = ', and not equal, '~=, are in a special class because they constrain their argument types not to a specific type but to be a set of four possible types, namely real, integer, char, or bool.

Their graph representation is:



Their type constraints are that T3 must be of type 'bool', that T1 and T2 must be of equivalent types, and that T1 and T2 must be oneof {bool, char, integer, real}.

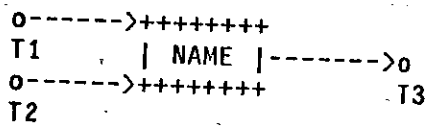
The action routine calls would be:

```
SetType (T3,bool)
SetMultType (T1,{bool,char,integer,real})
SetMultType (T2,{bool,char,integer,real})
equiv (T1,T2)
```

10.3 Boolean operations

Their are two classes of boolean operations in VAL. The first class has two arguments, the second has one.

The members of the class with two arguments are *and*, '&', and *or*, '|'. Their graph representation is:



```
o------>++++++++
T1       | NAME |------->o
o------>++++++++         T3
T2
```

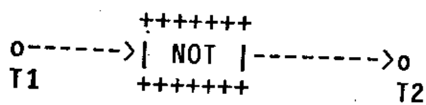
The type constraints are that T1, T2, and T3 are all of type 'bool'.

The action routine calls would be:

```
SetType(T1,bool)
SetType(T2,bool)
SetType(T3,bool)
```

The second class has only one member, the *not*, '~', operator.

Its graph representation is:



```
           +++++++
o------>|  NOT  |-------->o
T1         +++++++          T2
```

Its type constraints are that T1, and T2 must be of type 'bool'.

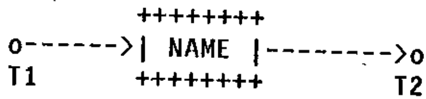
The action routine calls are:

```
SetType(T1,bool)
SetType(T2,bool)
```

10.4 Type Conversion Operations

Their are three operations intended to convert one data type into another. These are *real*, *character*, and *integer*.

Their graph representation is:



If the name is 'real' then the constraints are that T1 is of type 'integer' and T2 is of type 'real'. The actions routine calls are SetType calls in the obvious manner (Henceforth, all obvious SetType's are omitted.) If the name is 'character', the constraints are that T1 is of type 'integer' and that T2 is of type 'char'. If the name is 'integer', the constraints are that T2 is of type 'integer' and T1 is oneof {real, char}. Its action routine calls would be:

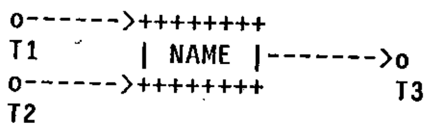
```
SetType (T2,integer)
SetMultType (T1,{real,char})
```

10.5 Real and Integer Operations

Most real and integer operations have the same names. Those that do are divided into four classes.

The first class takes 2 arguments and returns 1 result, all three types being the same type. The members of this class are *plus*, ' + '; *minus*, '-'; *multiply*, '*'; *divide*, '/'; *max*; and *min*.

Their graph representation is:

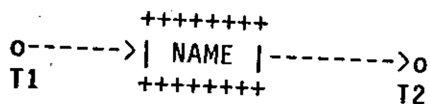


The type constraints are that all three types are constrained to be equivalent, and that each type is constrained to be oneof {real, integer}. The action routine calls would be:

```
SetMultType (T1,{real,integer})
equiv (T1,T2)
equiv (T1,T3)
```

The next class has one argument and one result, both of which are either real or integer types. The members of this class are *negation*, '-'; and *abs*.

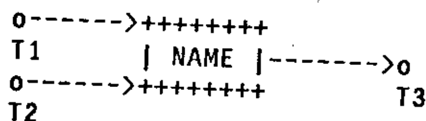
Their graph representation is:



The type constraints are that T1 and T2 are equivalent types and that they each must be oneof {real, integer}. The action routine calls would be:

```
SetMultType (T1,{real,integer})
equiv (T1,T2)
```

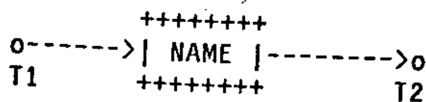
The next class has 2 arguments and 1 result, with the result being forced to be of type 'bool'. The members of this class >, <, >=, and <=. Their graph representation is:



The type constraints are that T3 is of type 'bool', that T1 and T2 must be of equivalent types, and that each of T1 and T2 may be oneof {real, integer}). The action routine calls would be:

```
SetMultType (T1,{real,integer})
equiv (T1,T2)
SetType (T3,bool)
```

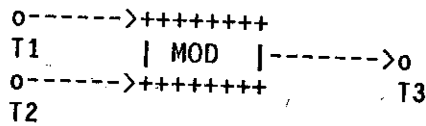
The fourth and final class of real/integer operations has 1 argument and 1 result. The argument can be real or integer, the result is a bool. The members of this class are *is pos-over*, *is neg-over*, *is unknown*, *is zero-divide*, *is over*, and *is arith-error*. Their graph representation is:



The type constraints are that T1 must be oneof {real, integer} and T2 must be of type 'bool'. The action routine calls would be:

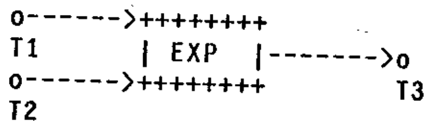
```
SetMultType (T1,{real,integer})
SetType (T2,bool)
```

Their are five operations that operate on real and integer types which do not fit into the above classes. The first of these special cases is *mod*. Its graph representation is:



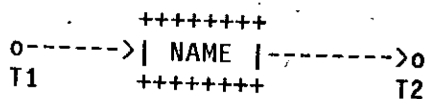
The type constraints are that T1, T2, and T3 must be of type 'integer'.

The second special case is *exp*. Its graph representation is:



The type constraints are that T1 and T3 must have equivalent types, that if T1 is known to be of type 'integer' or if T2 is known to be of type 'real' then T1 and T2 must have equivalent types. Otherwise, T1 and T2 must be oneof {real,integer}. Note, the action routine calls are omitted. Exp is a very nasty operator. I am in favor of reducing its possible arguments to both reals or both integers. It could then be treated as if it were in the first class of real/integer operations. No power is lost, since if you wanted to raise a real type to an integer power, you could just convert the integer to a real and use the real to the real case.

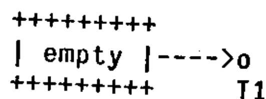
The final three special cases are *is pos-under*, *is neg-under* and *is under*. Their graph representation is:



The type constraints are that T1 is of type 'real' and that T2 is of type 'bool'.

10.6 The empty operation

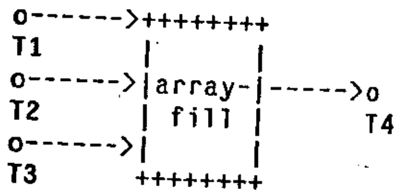
The graph representation of the *empty* operation is:



The type constraint is that T1 must be type 'empty'.

10.7 Array Operations

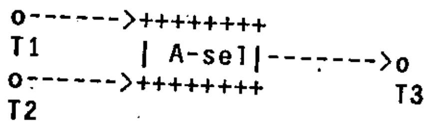
The graph representation for the operation *array-fill* is:



The type constraint are that T1 and T2 must be of type 'integer' and that T4 is an array of type T3. The action routine calls are:

```
SetType (T1,integer)
SetType (T2,integer)
merge (T4, T0) ; T0 := [array ( (T3) () )   ()          ={}  ]
                              sub-types  sub-fields  eq-list
```

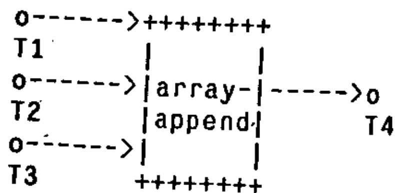
The graph representation for the [] (select) operation is:



The type constraints are that T1 is an integer and that T2 is an array of T3. The action routine calls are:

```
SetType (T1,integer)
merge (T2, T0) ; T0 := [array ( (T3) () )   ()          ={}  ]
                              sub-types  sub-fields  eq-list
```

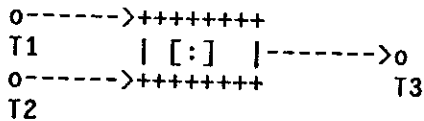
The graph representation of the *append* operation is:



The type constraints are that T2 is of type 'integer', that T1 and T4 are equivalent types and that T1 is an array of T3. The action routine calls are:

```
SetType (T2,integer)
equiv (T1, T4)
merge (T1, T0) ; T0 := [array ( (T3) () )    ()          ={}  ]
                                  sub-types  sub-fields  eq-list
```

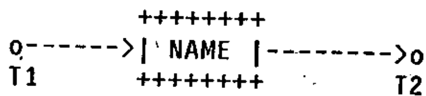
The graph representation for [:] (create by elements) is:



The type constraints are T1 is of type 'integer' and that T3 is an array of T2. The action routine calls are:

```
SetType (T1,integer)
merge (T3, T0) ; T0 := [array ( (T2) () )    ()          ={}  ]
                                  sub-types  sub-fields  eq-list
```

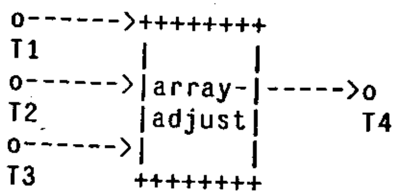
The following three operations have the same constraints; *array-limh*, *array-liml* and *array-size*. Their graph representation is:



The type constraints are that T2 is of type integer and that T1 is an array of any. The action routine calls are:

```
SetType (T2,integer)
merge (T1, T0) ; T0 := [array ( (Tc) () )    ()          ={}  ]
                                  sub-types  sub-fields  eq-list
Tc := any
```

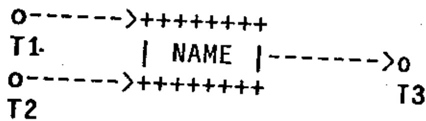
The operation *array-adjust* has the following graph representation:



The type constraints are that T2 and T3 are of type 'integer', that T1 and T4 are equivalent types and that T1 is an array of 'any'. The action routine calls are:

```
SetType (T2,integer)
SetType (T3,integer)
equiv (T1,T4)
merge (T1, T0) ; T0 := [array ( (Tc) () )    ()          ={}  ]
                                   sub-types  sub-fields  eq-list
Tc := any
```

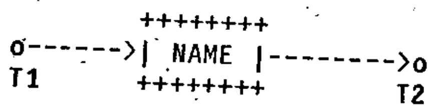
The operations *array-addh* and *array-addl* have the following graph representation:



The type constraints are that T2 must be of type 'integer' , that T1 and T3 are equivalent types, and that T1 is an array of any. The action routine calls are:

```
SetType (T2,integer)
equiv (T1,T3)
merge (T1, T0) ; T0 := [array ( (Tc) () )    ()          ={}  ]
                                   sub-types  sub-fields  eq-list
Tc := any
```

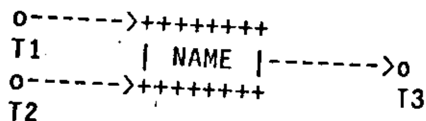
The operations *array-setl*, *array-remh* and *array-reml* have the graph representation:



The type constraints are that T1 and T2 are constrained to be equivalent, and that T1 is an array of any. The action routine calls are:

```
equiv (T1,T2)
merge (T1, T0) ; T0 := [array ( (Tc) () )    ()          ={}  ]
                                   sub-types  sub-fields  eq-list
Tc := any
```

The operations *concatenate*, '||'; and *array-join* have the graph representation:

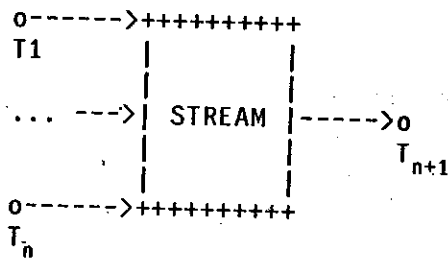


The type constraints are that T1 is equivalent to T2 and to T3, and that T1 is an array of any. The action routine calls are:

```
equiv (T1,T2)
equiv (T1,T3)
merge (T1, T0) ; T0 := [array ( (Tc) () )     ()         ={}  ]
                                 sub-types  sub-fields  eq-list
Tc := any
----------------------------------------------------------------
```

10.8 Stream Operations

The operation *stream* has the graph representation:

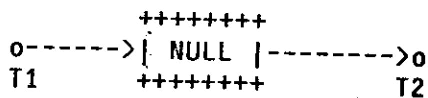


```
o------>++++++++++
T1      |        |
        |        |
... --->| STREAM |----->o
        |        |      T(n+1)
        |        |
o------>++++++++++
Tn
```

The type constraints are that T1 through T_n are equivalent and that T_{n+1} is a stream of T1. The action routine calls are:

```
equiv (T1, T2)
equiv (T2, T3)
...
equiv (T(N-1),T(n))
merge (T(n+1), T0) ; T0 := [stream ( (T1) () )    ()         ={}  ]
                                     sub-types  sub-fields  eq-list
```

The operation *null* has the graph representation:

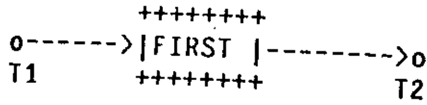


```
        ++++++++
o------>| NULL |-------->o
T1      ++++++++         T2
```

The type constraints are that T2 must be of type 'bool' and that T1 is a stream of any. The action routine calls are:

```
SetType (T2,bool)
merge (T1, T0) ; T0 := [stream ( (Tc) () )    ()         ={}  ]
                                 sub-types  sub-fields  eq-list
Tc := any
```

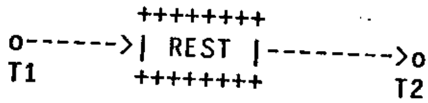
The operation *first* has the graph representation:



The type constraint is that T1 must be a stream of T2. The action routine call is:

```
merge (T1, T0) ; T0 := [stream ( (T2) () )     ()          ={}  ]
                                sub-types  sub-fields  eq-list
```

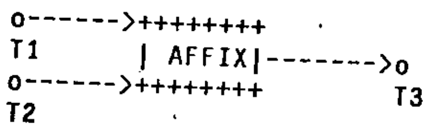
The operation *rest* has the graph representation:



The type constraints are that T1 and T2 are equivalent types and that T1 is a stream of any. The action routine calls are:

```
equiv (T1, T2)
merge (T1, T0) ; T0 := [stream ( (Tc) () )     ()          ={}  ]
Tc := any                       sub-types  sub-fields  eq-list
```

The operation *affix* has the graph representation:

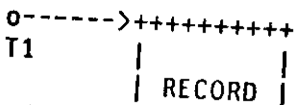


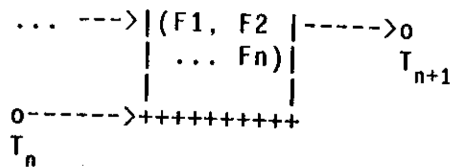
The type constraints are that T2 and T3 are equivalent, and that T2 is a stream of T1. The action routine calls are:

```
equiv (T2, T3)
merge (T2, T0) ; T0 := [stream ( (T1) () )     ()          ={}  ]
                                sub-types  sub-fields  eq-list
```

10.9 Record operations

The operation *record* has the graph representation:

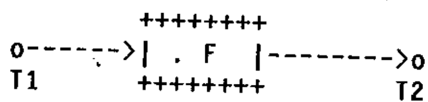




The type constraint is that T_{n+1} is a record for which F,T are the field name, sub-type pairs. The action routine call is:

```
merge (Tn+1, T0) ; where
T0 := {record} ( (T1 T2 ... Tn) () ) ( F1 F2 ... Fn ) ={}
```

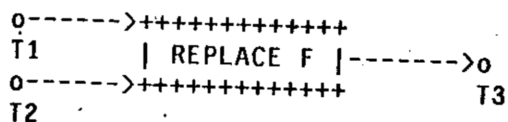
The operation *select*, '.', has the graph representation:



The type constraint is that T1 is a record with a fieldname,sub-type pair F,T2. The action routine call is:

```
merge (T1, T0) ; where
T0 := {record} ( (T2) () ) (F) ={}
```

The operation *replace* has the graph representation:

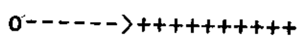


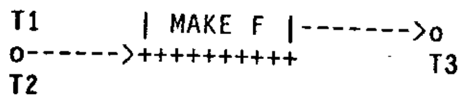
The type constraints are that T1 is a record with a fieldname, sub-type pair F, 'any' and that T3 is a record with a fieldname, sub-type pair F, T2. The action routine calls are:

```
merge (T1, T0) ; where
T0 := {record} ( (Tc) () ) (F) ={}
Tc := {any}
merge (T3, T0) ; where
T0 := {record} ( (T2) () ) (F) ={}
```

10.10 Union Types

The operation make has the graph representation:

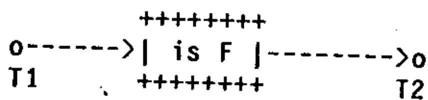




The type constraints are that T1 and T3 are equivalent, and that T1 is a oneof with fieldname, sub-type pair F, T2. The action routine calls are:

```
equiv (T1, T3)
merge (T1, T0) ; where
T0 := {oneof} ( (T2) () ) (F) ={}
```

The operation *is* has the graph representation:

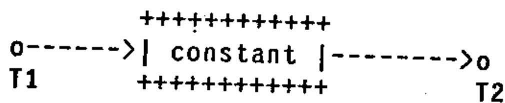


The type constraints are that T1 must be a oneof and that T2 must be of type 'bool'. The action routine calls are:

```
SetType (T2,bool)
merge (T1, T0) ; where
T0 := {oneof} ( () () ) () ={}
```

10.11 Constants

All constants have the graph representation:



The type constraint is that T2 must be the same type as the constant.

11 Basic Constructs

11.1 If Then Else constructs

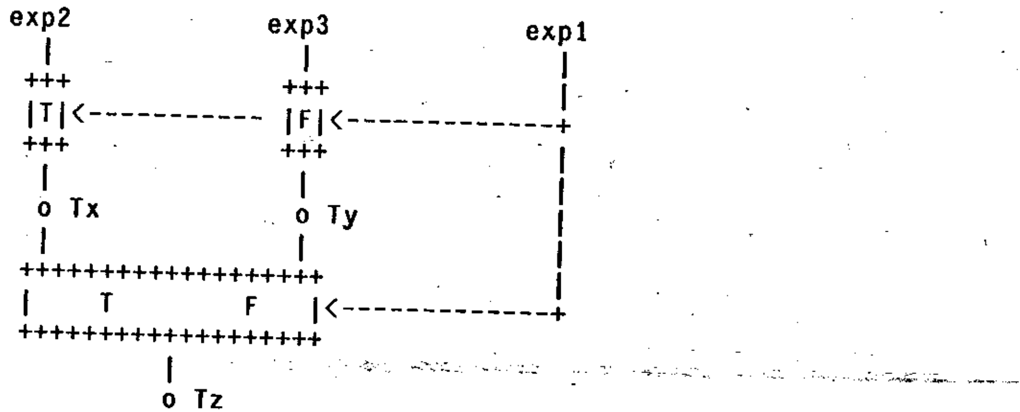
The *If Then Else* construct can be written in the form:

```
IF exp1 THEN exp2
```

```
ELSE exp3
ENDIF
```

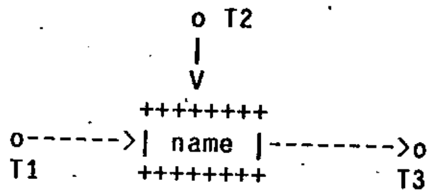
where exp1, exp2, and exp3 are all arbitrary VAL expressions.

The graph representation of the construct is:



In the graph fragment above, exp1, 2 and 3 are the graph representations of the VAL expressions. There are three operator nodes introduced above, the T node, the F node and the TF node.

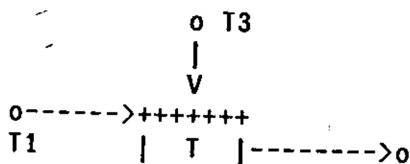
The *T* and *F* operator nodes have the graph representation:

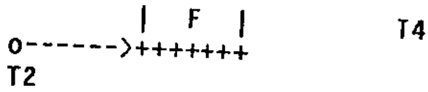


The type constraints are that T1 and T3 are equivalent and that T2 must be of type 'bool'. The action routine calls are:

```
equiv (T1, T3)
SetType (T2, bool)
```

The *TF* node has the graph representation:





The type constraints are that T1, T2 and T4 are equivalent, and that T2 is of type 'bool'. The action routine calls are:

```
equiv (T1, T2)
equiv (T1, T4)
SetType (T2, bool)
```

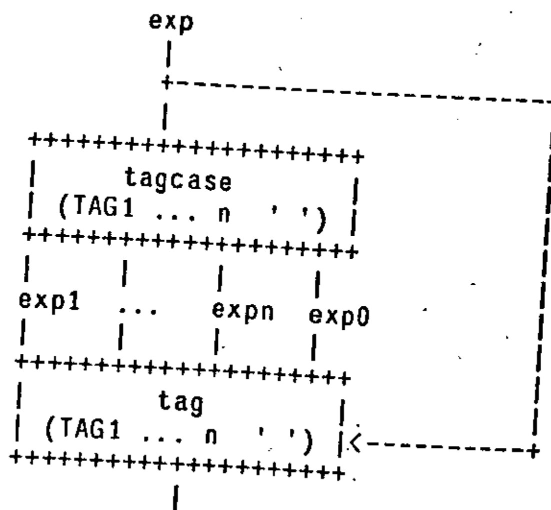
11.2 Tagcase constructs

The *tagcase* construct can be written in the form:

```
TAGCASE exp
        TAG1 : exp1
        TAG2 : exp2
          ...
        TAGn : expn
        OTHERWISE : exp0
        ENDTAG
```

where exp, exp0 to n are arbitrary VAL expressions.

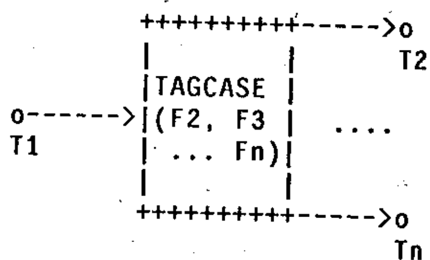
The graph representation of the construct is:



In the above graph fragment, all the exp's are short for the actual graph representation of that

expression. Note that the 'OTHERWISE' clause in the tagcase is represented as a ' ' (a blank). Two new operator nodes were introduced, tagcase and tag.

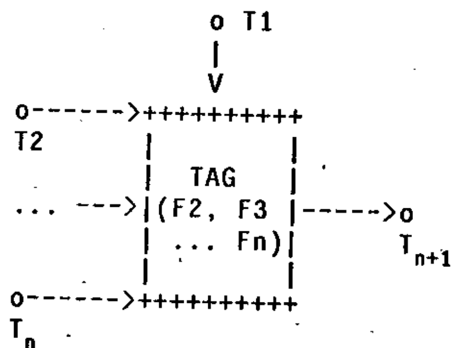
The *tagcase* node's graph representation is:



The type constraint is that T1 is a oneof with fieldname, sub-type pairs F2,T2 to Fn,Tn. The action routine call is:

```
merge (T1, T0) ; where
T0 := {oneof} ( (T2 T3 .... Tn) () ) ( F2 F3 ... Fn ) ={}
```

The *tag* node's graph representation is:



The type constraints are that T1 is a oneof with field F2 to Fn, and that T2 to T_{n+1} are equivalent. The action routine calls are:

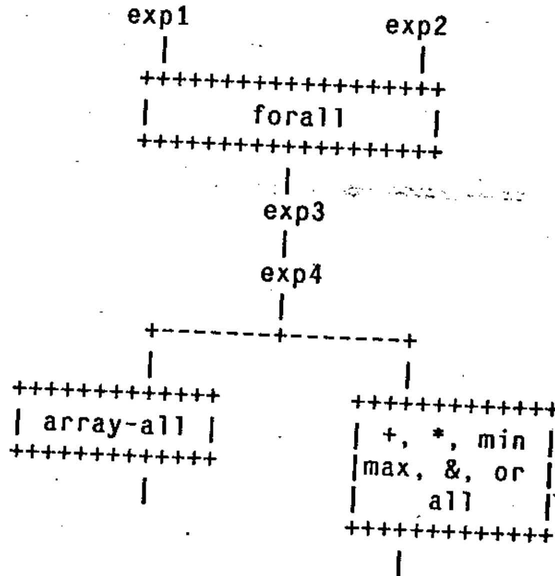
```
merge (T1, T0) ; where
T0 := {oneof} ( (Ta2 Ta3 ... Tan) () ) ( F2 F3 ... Fn) {}
Ta2, Ta3, ..., Tan := {any}
equiv (T2, T3)
equiv (T3, T4)
...
equiv (Tn,Tn+1)
```

11.3 Forall constructs

The *forall* construct can be written in the form:

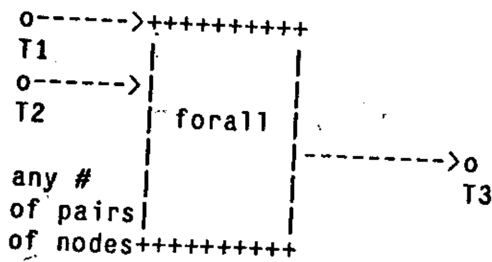
```
FORALL var in [exp1, exp2]
        exp3
        CONSTRUCT or EVAL exp4
        ENDALL
```

Note that in the following graph representation var is used to tie the result type of the forall node into exp3 and exp4. This is not shown as it is inherent in the links.



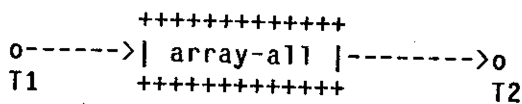
The array-all node represents the construct operation. The other all node represents all the possible eval operations.

The graph representation of the *forall* node is:



The type constraints is that all type nodes must be of type integer. Action routine calls are all SetType (T, integer).

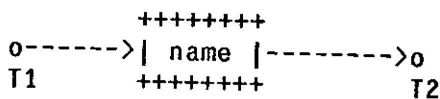
The graph representation of the *array-all* node is:



The type constraints is that T2 is an array of T1. The action routine call is:

```
merge (T2, T0) ; T0 := [array ( (T1) () )    ()         ={}  ]
                                sub-types  sub-fields  eq-list
```

All the eval nodes have the graph representation:



The operations *plus-all*, *times-all*, *min-all* and *max-all* are all treated as if they were real/integer operations of the second class. The operations *or-all* and *and-all* constrain T1 and T2 to be of type 'bool. The action routine calls are both SetType (T, bool).

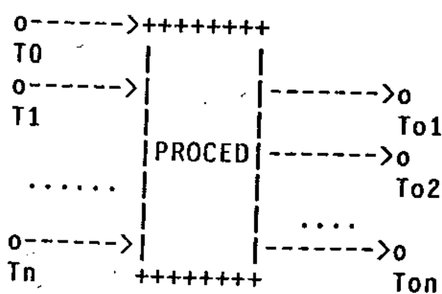
----

12 Functions

There are two ways a function type is encountered in VAL. The first is in a function header and the second is when the function is referenced. Consider the first case:

```
function foo (x1, ..., xn)
```

The graph representation is:

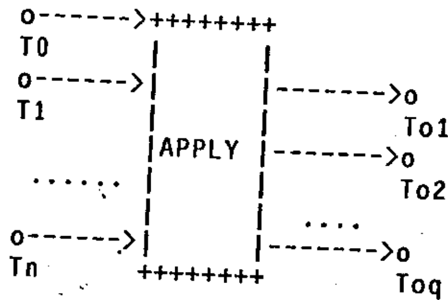


Here foo and x1 to xn are placed in the symbol table with foo pointing at T0 and x1 to xn at T1 to Tn. The type constraints are that T0 is a function with arguments T1 to Tn, and that T1 is

equivalent to To1, T2 to To2 and so on. The action routine calls are:

```
merge (T0, Tc) ; where
Tc := {function} ( (T1 T2 ... Tn) () ) () ={}
equiv (T1, To1)
equiv (T2, To2)
  .....
equiv (Tn, Ton)
```

The second case is of the form foo (x1, ..., xn). Its graph representation is :



Note the result type nodes need not have the same arity as T1 to Tn. Note also that only foo is placed into the symbol table. If it was there already a *link* node (see below) is used to connect this reference to foo with the proced or apply node that cause it to be in the symbol table already. If it was not in the table, foo will point to T0.

The type constraints are that T0 is constrained to be a function with argument sub-types T1 to Tn and result sub-types To1 to Toq. The action routine call is:

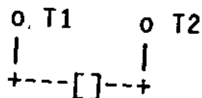
```
merge (T0, Tc) ; where
Tc := {function} ( (T1 T2 ... Tn) (To1 To2 ... Toq) ) () ={}
```

---

13 Special Nodes

There are two special nodes that are used to help construct the data type graph. These are the *merge* node and the *link* node.

The graph representation of the *merge* node is:

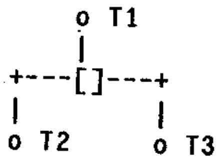


```
       |
       o T3
```

The type constraints are that T1, T2 and T3 are all equivalent. The action routine calls are:

```
equiv (T1, T2)
equiv (T1, T3)
```

The graph representation of the *link* node is:



The type constraints are that T1, T2, and T3 are all equivalent. The action routine calls are:

```
equiv (T1, T2)
equiv (T1, T3)
```

References

[Ack 79]

Ackerman, William B. and Jack B. Dennis.
VAL -- A Value-Oriented Algorithmic Language: Preliminary Reference Manual.
1979.
MIT/TR - 218

[Dennis 66]

Dennis, Jack B. and van Horn.
Programming Semantics for Multiprogrammed Computations.
Communications of the ACM 9 :143-155, March, 1966.

[Dennis 75]

Dennis, Jack B.
First Version of a Data Flow Procedure Language.
Technical Memoranda, Massachusetts Institute of Technology, May, 1975.

[Dennis 81]

Dennis, Jack B.
Data Should Not Change: A Model for a Computer System.
Technical Memoranda, Massachusetts Institute of Technology, July, 1981.

[Kral 73]

Kral, J.
The equivalence of modes and the equivalence of finite automata.
Algol Bulletin 35:34-35, March, 1973.

[Weng 75]

Weng, Kung-Song.
Stream-Oriented Computation in Recursive Data Flow Schemas.
Technical Memoranda, Massachusetts Institute of Technology, October, 1975.

[Weng 80]

Weng, Kung-Song.
An Abstract Implementation For a Generalized Data Flow Language.
Technical Report, Massachusetts Institute of Technology, February, 1980.

[Zosel 71]

Zosel, M. E.
A Formal Grammar for the Representation of Modes and its Application to Algol 68.
Technical Report, Computer Science Group, University of Washington, 1971.