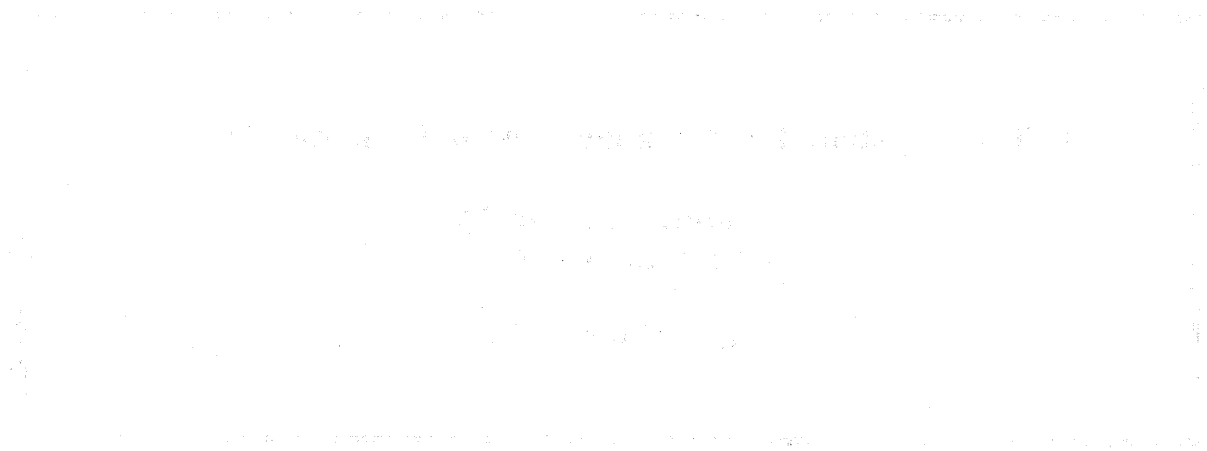
A Type System for Languages with Dynamic Arrays

Thomas M. Derby
Benjamin G. Zorn

CU-CS-874-99

University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE



ANY OPINIONS, FINDINGS, AND CONCLUSIONS OR RECOMMENDATIONS EXPRESSED IN THIS PUBLICATION ARE THOSE OF THE AUTHOR(S) AND DO NOT NECESSARILY REFLECT THE VIEWS OF THE AGENCIES NAMED IN THE ACKNOWLEDGMENTS SECTION.

A Type System for Languages with Dynamic Arrays

Thomas M. Derby and Benjamin G. Zorn

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309–0430 USA

CU-CS-874-98 November 1998

University of Colorado at Boulder

Technical Report CU-CS-874-98
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

Copyright © 1999 by
Thomas M. Derby and Benjamin G. Zorn

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309–0430 USA

A Type System for Languages with with Dynamic Arrays*

Thomas M. Derby and Benjamin G. Zorn

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309–0430 USA

November 1998

Abstract

A common coding error made when using languages for prototyping numerical computations (such as Matlab) is to have a dimensionality or matrix size conflict. Such errors are typically only detected at runtime in such languages because no notion of array size is included as part of the static type checking system (if one is provided at all). Therefore, these errors are only caught when the matrix expression is actually evaluated, and the result is that errors are often reported from library code, or other routines not written directly by the user. This paper presents a scheme for adding size declarations for array arguments and inferring information about the sizes of intermediate variables in a simplified array-based imperative language. It defines an entity called the *abstract extent graph* and describes how to use it to perform static size type checking using a solution method similar to a common technique for minimizing finite state machines. The resulting algorithm requires $O(s^2v^2)$ for each procedure in the worst case, where s is the size of the procedure or function, and v is the number of variables used in it.

Research supported by NSF Grant ASC-9307315.

1 Introduction

The tension between static and dynamic type checking can lead to a difficult design decision, particularly in languages for prototyping scientific computations. Unfortunately, static checking of array sizing is typically not available, despite the general acceptance of static type checking as a valuable programming tool. One reason for this is that the size of the array often needs to be a runtime-computed quantity, and therefore attempting to check the exact sizes of arrays at compile-time is an undecidable proposition. Sophisticated techniques to deduce sizing information could be used, but because they are heuristic in nature, they are not ideal for use as a typing system (although perfectly suitable for compiler optimization purposes).

Rather than defining an operation to be legal if the runtime sizes are compatible, however, we can define a type system which is somewhat more restrictive than this, making it fully checkable at compile time. In order for such a system to be effective, of course, it must provide enough flexibility that typical "runtime type-legal" programs will still typecheck correctly. If this could be done, as long as there is a reasonable way to handle cases in which the size checking does fail, the benefits of such a system would seem to outweigh its costs.

Our research into EQ, a new language for prototyping scientific computations [4, 5, 6] has lead us to investigate typing systems that include sizing information. This paper is an exploration of one such system, applied to a simplified array operation-based imperative language similar to Matlab. The type system is based on the notion of an *abstract extent* (in contrast with the exact size, which is a real extent).

2 Motivation

In order to motivate creating a type system that can handle array size as part of its type, consider the following procedure:

```
proc mat_mat_multiply (in x, in y, out z)
```

This signature tells us relatively little about when it is legal to use this procedure, and when it is not. In addition to providing little help to the user to understand what the routine is intended to do, it also can cause error propagation; the runtime error message might not be returned directly from `mat_mat_multiply`, but from one of the procedures called by it. In fact, this "error propagation" could propagate arbitrarily far down the calling chain. The result is error messages which only occur at runtime, and are very difficult to decipher.

In order to combat this problem (among others), standard languages add types to the argument declarations. Languages such as Pascal have fixed size arrays, so that our declaration becomes

```
proc mat_mat_multiply (in x[1..100,1..100], in y[1..100,1..100],
  out z[1..100,1..100])
```

This system treats array size as part of the type, and avoids the problems mentioned above. However, this system is not flexible enough for many real programs, because it cannot support varying-sized arguments. One of the features that has allowed Fortran to endure in the scientific world is its support for flexible-sized procedure arguments. But Fortran cannot type-check its procedure arguments, because the size is fully runtime determined, and is therefore not part of the type system.

The signature that we really want to write is something closer to

```
proc mat_mat_multiply (in x[R,S], in y[S,T], out z[R,T])
```

Here, we've used additional names to indicate the size of each array in symbolic fashion. The relationships between the three argument's sizes have been completely captured. We refer to the identifiers `R`, `S`, and `T` as *abstract extents*. Using this signature, we can check size compatibility for calls to this routine if the actual argument variables also have abstract extents[1]

```
decl a[M,N];
decl b[N,O];
decl c[M,O];

mat_mat_multiply (a, b, c);      ==> Legal
mat_mat_multiply (a, c, b);      ==> Illegal
```

Note that we always treat two different abstract extents (such as `M` and `N` above) as different, even though they might be the same for a particular context. For example, the code

```
decl x[S];
decl z[T];

if (S == T)
  z = z + x;
```

will not typecheck.[2] This code would be legal in a type system based on exact sizes (runtime or otherwise). But in our abstract extent system, the statement `z = z + x` is a type mismatch because `x` and `z` do not have the same abstract extents. It is this simplification that makes size checking based on abstract extents possible to perform efficiently at compile time.

Clearly, declaring all of the sizes of the variables in a procedure will allow it to be typechecked. However, doing so would quickly become cumbersome for array intermediates. Therefore, the system presented in this paper requires complete procedure signatures, but deduces the sizes of intermediate variables.

[1] In our actual system, the extents of the actual parameters will be automatically deduced. We use `decl` statements here to simplify the presentation.

[2] We note that `decl` and `==` do not appear in the simplified language presented in the main body of the paper. They are used here, and in other illustrative examples, to simplify the presentation.

3 Existing Type Systems

3.1 Fortran and Fortran90

Although Fortran [1, 12] is one of the earliest languages still in widespread use today, it has some of the most useful features for handling array sizes of the languages mentioned in this section. While not very strong on checking the sizes of operations, it is very good at *expressing* size relationships. Arrays are passed to subroutines by reference, and their size is declared within the subroutine. In order to do this, the user must also pass along integer parameters indicating the size of the arrays.

```
SUBROUTINE MATVEC(MAT, VEC, OUTPUT, N, M)
DIMENSION MAT(M,N)
DIMENSION VEC(N)
DIMENSION OUTPUT(M)
....
END
```

This permits the programmer to indicate the required size relationships between the arguments, but does not enforce these relationships at compile time. Further, since the size parameter is not formally related to the arrays themselves, a programmer can pass the wrong size parameter, producing highly unpredictable results.

3.2 Ada

Ada [3] has a very rich and complex type system. The property of interest to us here is its ability to allow unconstrained array types, which permits a call to a procedure or function with an argument whose exact size is not known at compile time:

```
type MATRIX is array(INTEGER range <>, INTEGER range <>) of REAL;
```

However, there is no way to express the relationships *between* parameters. Thus, the best declaration we can create for a matrix-vector multiply routine is

```
type MATRIX is array(INTEGER range <>, INTEGER range <>) of REAL;
type VECTOR is array(INTEGER range <>) of REAL;
procedure MAT_VEC_MULTIPLY(M : in MATRIX, V : in VECTOR,
                           R : out VECTOR);
```

Checking the sizes of the arrays for compatibility is quite possible, and would prevent any possibility for user error in specifying the size of the arguments. However, the checking must be performed at runtime, which not only reduces performance but also leaves open the possibility of "error propagation", as discussed earlier, if runtime size checking is not strictly performed for all program sections.

3.3 Matlab

Matlab [15] uses a fully runtime typing system, which allows a programmer to write routines that take variable sized objects, but errors in their shape are only caught at runtime. Worse, these errors may be detected in a lower-level library routine called by the errant code. The result is errors that can be very difficult to track down and fix. Since Matlab does not employ a user-level typing system, the type signature for our example is very simple to express:

```
Z = mat_vec_multiply (M, V)
```

However, runtime typing systems do not offer the advantages of a compile-time type checker.

3.4 Haskell

Haskell [11][10] is an example of a Hindley-Milner type system, extended to handle type (and recently constructor) classes. For example, a typical type signature for matrix vector multiplication in Haskell looks like

```
matvec :: (Ix a, Ix b, Num c) =>
          Array (a,b) c -> Array (b) c -> Array (a) c
```

where `a` and `b` represent the types of the indices, and `c` represents the type of the result. Unfortunately, despite the fact that this code *looks* like it is matching dimension lengths, it is not; the size of a dimension is not part of the index type. The matching of `a` and `b` only ensures that the type of the indices (character, integer, or boolean, for example) correspond correctly.

4 Other Related Work

In addition to other type systems used in programming languages, there are a number of other areas of computer science that are related to the work described in this paper. We briefly mention some of these below.

4.1 Compiler Optimization Analysis of Subscript Ranges

Many compiler optimizations depend on determining the possible range of values for array subscript expressions [2, 14]. By knowing the extent of the possible values for a subscript, a compiler may be able to omit runtime range checking, as well as perform other optimizations and detect parallelism. Since these techniques focus on the ranges of scalar variables, rather than the size of variables, and because they are heuristic in nature, they cannot be directly applied to type deduction problems.

```
Program -> Proc_1 ... Proc_n              Programs

Proc -> proc Name (ArgList) Stmt          User-defined procedure
      | proc Name (ArgList) builtin       Builtin procedure declaration

ArgList -> Arg_1, ..., Arg_k
Arg -> Direction Name Sizing
Direction -> in
           | out
           | in out
Sizing -> <nothing>                       Scalar
        | [Name_1, ..., Name_p]           List of Extents

Stmt -> Name (Name_1, ..., Name_m)        Procedure Call
      | if (Name) Stmt else Stmt          Conditional construct
      | while (Name) do Stmt              Looping construct
      | {Stmt_1; ...; Stmt_l}             Compound statement
```

Figure 1: A Compile-time Typed Language with Arrays

4.2 Abstract Interpretation

One standard (and very useful) framework for computing "limited" information about the runtime behavior of a program at compile time is known as abstract interpretation [7, 13]. An abstract interpretation amounts to a mapping from some aspect of a program's behavior onto a lattice structure.

Unfortunately, if the required information about the program (in our cases, size relationships) cannot be view in terms of a lattice, this technique does not apply. And if the lattice does not have finite height (as we believe must be the case for our sizing system), then approximate methods of solution are typically used—which places us back into the heuristic category, with all the same problems previously mentioned.

5 Language Extensions to Support Compile-Time Sizing

In a "standard" interpreted prototyping language such as Matlab, the most "restrictive" type signature we could give a matrix-vector multiply would be something like

```
proc matvec (out result, in m, in v)
```

Our basic goal is to add to this language enough machinery to be able to express the sizing relationships between variables and parameters. We do this by introducing a new user-level construct, the *extent*. An extent represents a range of integers to be used as the limits for a particular dimension of an array. (Note: we adopt the convention in this paper of capitalizing these identifiers in our examples.) We can now extend the procedure declaration syntax of our language, and express the desired procedure signature as

```
proc matvec (out result[S], in m[S,T], in v[T])
```

This signature says that the size of v must be equal to the size of the second dimension of m, and that the result will have the same size as the first dimension of m. Also note that the dimensionality of each argument has also been specified implicitly. The syntax of the resulting language is given in Figure 1.

Because no expressions exist, operators such as + and - must be coded using built-in procedures. For example, the statement a = b + c * d[i,j] might be translated into the following code:

```
matrix_subscript (temporary1, d, i, j);
scalar_multiply (temporary2, c, temporary1);
scalar_add (a, b, temporary2);
```

We give code for a matrix-vector multiplication routine in Figure 2, which will serve as a primary example throughout the text.

6 Informal Semantics of the Typed Language

Each procedure is typed separately, needing only the signatures of the procedures it calls to complete its type checking. This ensures that recursive procedures pose no problems to the typing algorithm, as all procedure signatures must be explicitly given. The typing assigns a dimension and set of sizes to each variable in the procedure at every lexical point. The rank (number of dimensions) of a variable is required to be constant throughout the procedure. These can be computed and checked for consistency using very standard data-flow algorithms (which we omit here).

We describe the relationship between the size information just before a given statement (the "input" sizes), and the size information just after a given statement (the "output" sizes) in the following sections.

6.1 Semantics of Procedure Calls

Procedure calls behave exactly as one might expect from their signatures. Input and "in out" arguments must have abstract sizes which obey the symbolic constraints of the procedure signature. Violations of this condition are a typing error.

The size of the output parameters' dimensions are determined as follows. First, if the symbolic extent referenced in the procedure's type signature for a particular dimension of an output parameter is provided by one of the inputs, then that size is used. Otherwise, the symbolic extent for a particular dimension of an output parameter is new, and a brand new abstract extent, different from all existing abstract extents, is created.

```
proc matvec_product (out result[S], in m[S,T], in v[T])

  /* Get the upper and lower bounds of m's first dimension */
  get_lower_bound_matrix_dimension_1 (l1, m);
  get_upper_bound_matrix_dimension_1 (u1, m);

  /* Create storage for our result matrix */
  get_lower_bound_matrix_dimension_2 (l2, m);
  extract_matrix_column (result, m, l2);
  clear_vector (result);

  /* Loop over the rows of m */
  scalar_copy (i, l1);
  is_less_than_or_equal_to (condition, i, u1);
  while (condition)

    /* Get the i'th row of m */
    extract_matrix_row (current_row, m, i);

    /* Use a vector-vector dot product to compute part of our answer */
    dot_product (answer, current_row, v);

    /* Store our answer */
    set_vector_element (result, i, answer);

    /* Move on to the next row */
    scalar_add (i, i, 1);

    /* Re-check our condition for loop termination */
    is_less_than_or_equal_to (condition, i, u1);


% Signatures for the subroutines called by this procedure
  proc get_lower_bound_matrix_dimension_1 (out dim, in m[R][S]);
  proc get_upper_bound_matrix_dimension_1 (out dim, in m[R][S]);
  proc get_lower_bound_matrix_dimensions_2 (out dim, in m[R][S]);
  proc extract_matrix_column (out result[R], in m[R][S], in index);
  proc clear_vector (in out v[R]);
  proc scalar_copy (out result, in value);
  proc is_less_than_or_equal_to (out result, in a, in b);
  proc extract_matrix_row (out result[S], in m[R][S], in index);
  proc dot_product (out result, in v1[S], in v2[S]);
  proc set_vector_element (in out result[S], in index, in value);
  proc scalar_add (out result, in a, in b);
```

Figure 2: A Matrix-Vector Multiplication Procedure in Language T

6.2 Semantics of a Compound Statement

The typing for a compound statement is generated by processing its components one at a time. The output abstract extents from one statement are provided as input to the next.

6.3 Semantics of Control Structures—If Statements

The abstract extents at the "then" and "else" branches are equal to the abstract extents on input to the `if` statement itself. Then, the two bodies can be type checked, yielding a set of output sizes for the bodies. The only catch is what to do when the sizing information from the two branches is not the same.

One approach would be to declare a typing error. However, this does not provide size flexibility (it logically leads to a system where an array must have the same size for its entire lifetime). Thus, we instead introduce new abstract extents for the conflicting sizes. However, we introduce only one new abstract extent for each possible conflict. Thus, for example, if the "then" and "else" branches return the sizes `x[R],y[S],z[S],t[S]` and `x[R],y[R],z[R],t[Q]` then the output sizes from the if statement will be `x[R],y[`τ_1`],z[`τ_1`],t[`τ_2`]` where τ_1 and τ_2 are new abstract extent objects.

6.4 Semantics of Control Structures—While Loops

Because abstract extents must be assigned statically, entry into a looping control structure can change the abstract extents of a variable. For example, consider the code fragment

```
while (...) {
  x = z;
}
```

If the input sizes to this code are

```
x[S], z[R]
```

the variable `x` has abstract extent `S` just before the loop. However, upon entering the loop, `x` must be given a brand new extent, because at this lexical point in the code `x` and `z` are not guaranteed to have the same size. Our typing system needs to do a minimal amount of extent changing (to ensure that as many runtime-size-correct programs will typecheck as possible).

The sizing of `while` loops in our method can be described in the following iterative fashion. First, we hypothesize that the size information at the start of the loop body is the same as the size information lexically just before the `while` loop. We then type the body, under this assumption. If the size information we get back is identical to the input sizes, then we have a correct typing. The output sizes for the `while` statement are the same as those for its body.

However, if the sizes do not agree, then just as for the `if` statement, we have to introduce new abstract extents for the conflicting sizes. In this case, however, this simply generates a new, better approximation to the input sizes to the body. We must retype the body under these new types, and check again. This process is repeated until the input and output sizes for the `while` loop body are equivalent up to a renaming of the abstract extents introduced in the solution process. The method is guaranteed to terminate because each iteration produces more distinct sizes than the previous one, and we must terminate once every variable has a distinct size from every other variable.

In our simple example above, just before the `while` loop the extent information is `(z[R], x[S])`. Hypothesizing this for the input sizes to the body, we get a set of output sizes of `(z[R], x[R])`. Since this is not equivalent to the input, we must introduce new abstract extents. We try again, with an input sizing set of `(z[R], x[`τ_1`])`. This again generates an output of `(z[R], x[R])`. However, this is compatible with `(z[R], x[`τ_1`])` (by substituting `R` for τ_1). Therefore, the typing is complete, and the output sizes from the while loop are `(z[R], x[R])`.

This is not a suitable technique for actually computing abstract extents, however, because the number of "hypothetical typings" performed can be exponential. This situation can occur if the body of a `while` contains further `while` loops. A much more efficient way of type checking programs in our sample language will be described in Section 8.

6.5 Semantics of Procedure Definitions

We now have described the abstract extent deduction process for statements. All that remains is to describe the relationship of procedure definitions to it. The sizes at the start of the procedure body are determined as follows. For variables which are `in` or `in out` parameters, their sizes are set to brand new abstract extents, except that sizes which are guaranteed to be equal based on the type signature have the same abstract extent. All other variables (including output parameters) are given the size $\perp$, which represents a fully unknown size. We can then check to make sure that the output abstract extents from the procedure body are compatible with the procedure's type signature.

7 The Abstract Extent Graph

The typing problem described above is somewhat difficult to solve because of the circular relationships. The "size generalizing" operations mean that it cannot be viewed as a finite lattice system. Further, approximations are not appropriate for a type system, which means that approximate solutions based on narrowing and widening operators (or any other technique) do not solve the problem. Therefore, we must develop an exact solution technique.

To investigate how to solve this problem, we define an *abstract extent graph*, which is a directed graph (V, E) where the arcs pointing towards a particular vertex are *ordered*. Each vertex v is labeled with a label

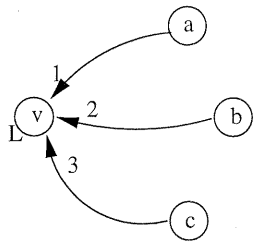


Figure 3: Typical Vertex in an Abstract Extent Graph

$c(v)$, which represents the *context* of that vertex. The edges are unlabeled, except for the ordering of all edges arriving at a particular vertex v. The vertices connected to a particular vertex v will be represented using the function $S_i(v)$, where i represents an integer based on the ordering of the vertices. A typical vertex is illustrated in Figure 3, along with it's context label and S_i relationships. We shall use the same notation for abstract extent graphs throughout the paper.

The intention is that each vertex v will represent a particular dimension of some variable at some location in the program (the size information and type of that variable can change depending on where in the program you are currently located). For example, a vertex might represent the size of the second dimension of program variable `X` after the 6th line of a given procedure. The locations are represented by the context labels $c(v)$; two vertices with the same context labels represent sizing information at the same point within the program. The solution process will assign to each vertex an *abstract extent* object representing the set of possible sizes that the described object can take, that obeys the following rule: if two vertices with the same context labels have the same abstract extent, then the size of the dimensions corresponding to these vertices will always be the same whenever the portion of the program represented by the context label is reached.

This information can be used within a type checker to make sure that the arguments to array operators and procedure calls have compatible types. Since we want to make as many programs legal as possible, we need to label the abstract extent graph with a *minimal* number of abstract extents.

7.1 Construction of the Abstract Extent Graph

An abstract extent graph is constructed in a structural way; each language statement corresponds to a graph piece which has "input" nodes for all variables used by the statement, and "output" nodes for all variables defined by the statement.

In addition, one extra vertex, written $\perp$, is added to the graph. This vertex represents the "undefined" size, and will be given a unique abstract extent (which will also be called $\perp$; the proper meaning can be determined by context). This vertex allows for the construction of "new" abstract extents for returns from procedures (why this construction works will be explained in Section 7.2).

The only primitive statement in our language is the procedure call. The abstract extent graph for this statement is constructed based on the procedure's type signature by the following intuitive rules:

1. Create one vertex for each extent name used in the procedure's signature. Label each with a brand-new, distinct context label.
2. Add arcs from the vertices representing the dimensions of the actual parameters just before the procedure call to the corresponding extent vertex created in step 1 for parameters which are `in` or `in out` parameters.
3. Add arcs from $\perp$ to any extent object created in step 1 which does not yet have any input arcs.
4. Add arcs from the new extent vertices to the vertices representing the dimensions of *all* parameters just after the procedure call.

Note that the order of the arcs pointing towards the new vertices is not important, as these vertices will only be used for error checking (see Section 7.3).

As an illustrative example, consider the procedure definition

```
proc mat_vec_multiply (in m[R,S], in v[S], out result[R]) {
  ...
}
```

and a call to that procedure

```
mat_vec_multiply (x, y, z);
```

Then, the portion of the abstract extent graph that is constructed for this procedure call is represented pictorially in Figure 4. The nodes labeled R and S represent the graph nodes that correspond with the abstract extents `R` and `S` from the procedure's type signature.

In order to be able to describe in a succinct fashion how to construct abstract extent graphs for control structure statements, we use a substitutive graphical system. In these diagrams, the inner rectangular boxes represent subgraphs generated by the sub-statements of a given statement (such as a conditional). The diagrams representing the child statements are then to be substituted in for the corresponding box in the parent statement's diagram. We follow the convention that the left side of the rectangular box has connections representing the sizes of all procedure variables before execution of the statement, and the right side has corresponding connections representing sizes on exit from the statement.

Variables which are not involved in a given statment (such as `P` in our example) will have arrows (one for each dimension) that simply traverse the box from left to right. Further, variables which are redefined within the statement may have input connections which do not connect to anything (as illustrated by `Z`).

The rules for the compound statement and `if` and `while` control constructs can now be described in this "boxed" representation, and are given in Figure 5 in pictorial form.

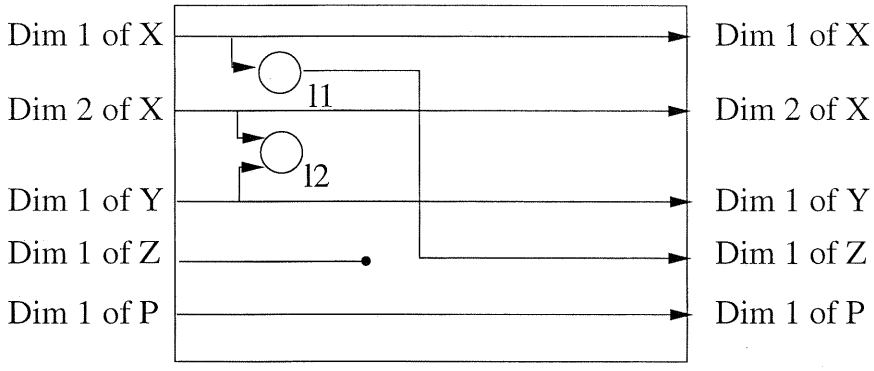


Figure 4: Example of Substitutive Graphical System

Finally, we need to be able to handle whole procedures. Given the abstract extent graph representing the body of a procedure, the graph to represent the whole procedure is constructed as follows:

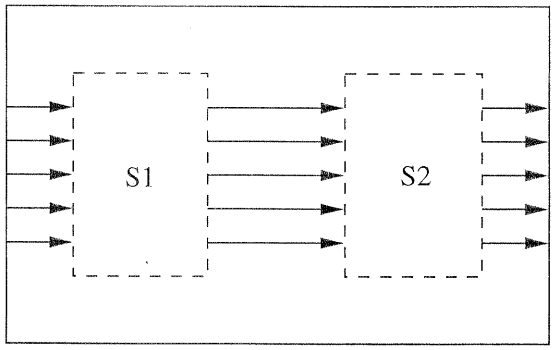
1. Create new vertices to represent each abstract extent used in the declarations for the formal parameters with direction `in` or `in out`. Label each with a brand new, unique context label. Add an arc from $\perp$ to each of these vertices.
2. Add arcs to the input vertices of the body's abstract extent graph from the corresponding abstract extent vertex created in step 1, for the `in` and `in out` parameters.
3. For all other variables used in the body, add arcs from $\perp$ to their input vertices.
4. Create new vertices to represent each abstract extent used in the declarations for the formal parameters with direction `in out` or `out`. Label each with a brand new, unique context label. Note that some abstract extents will now have *two* vertices associated with them.
5. Add arcs from the output vertices of the body's abstract extent graph to the corresponding extent vertices created in step 4, for `in out` and `out` parameters.
6. For any abstract extent which was given two vertices, add an arc from the one created in step 1 to the one created in step 4.

We produce one such graph for each procedure in the program. Each procedure can be type checked independently, because the complete procedure signatures are given for every procedure.

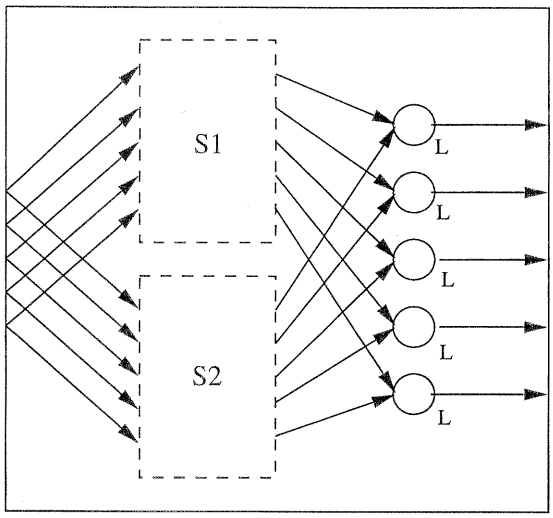
We give, in Figure 6, the abstract extent graph corresponding to our matrix-vector multiplication example procedure of Figure 2.

7.2 Characterizing a Solution of the Graph

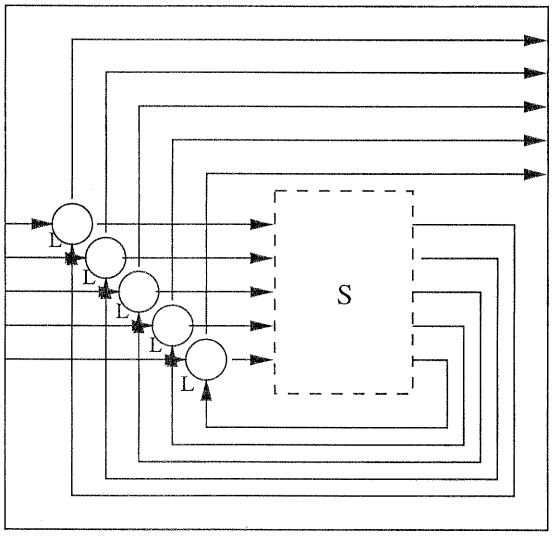
By a *solution* of an abstract extent graph (V, E), we mean an association of an abstract extent object with each node of the graph (represented by the function $\tau(V)$). Letting $S_i(x)$ represent the ith vertex connected to vertex x (recall that the arcs pointing to a particular vertex are ordered), we can express the requirements on function τ by the following relation:



S1; S2 (Sequential Execution)



if C S1 S2 (Conditional Statement)



while C do S (While Loop)

Figure 5: Rules for Building the Abstract Extent Graph for Control Structures

$$\tau(x) = \tau(y) \Rightarrow uniform(x) \ \vee \ uniform(y) \ \vee \ (c(x) = c(y) \ \wedge \ \forall_i \tau(S_i(x)) = \tau(S_i(y)))$$
$$\text{where } uniform(x) \equiv \forall i : \tau(S_i(x)) = \tau(x)$$

Informally, $uniform(x)$ indicates that all immediate ancestors of x have the same size as x does, which means that no "size generalization" is being performed. The rule as a whole requires that if x and y have the same abstract extent, then either they come from the same control structure and have the same sized inputs from each control path, or one of them is a combination of abstracts extents that are all the same.

This relationship, however, is satisfied by giving all vertices the same extent. We therefore add a second restriction:

$$\tau(x) = \tau(\perp) \Rightarrow x = \perp$$

which implies that $\perp$ has a different extent than all other vertices. In turn, this implies that all of the "new extents" we created by making an arc from $\perp$ and using a distinct context label will actually be distinct (which justifies using $\perp$ to construct new abstract extents, as was done in Section 7.1).

These relationships permit multiple possible functions τ that are solutions. Since we want as many programs to be type-legal as possible, what is desired is clearly the solution with a *minimum* number of abstract extents (we prove that such a solution exists and is unique in the process of proving our algorithm for computing it correct; see Section 8). An abstract extent labeling for our example matrix-vector abstract extent graph is given in Figure 6 (the numbers within the vertices represent the equivalence classes).

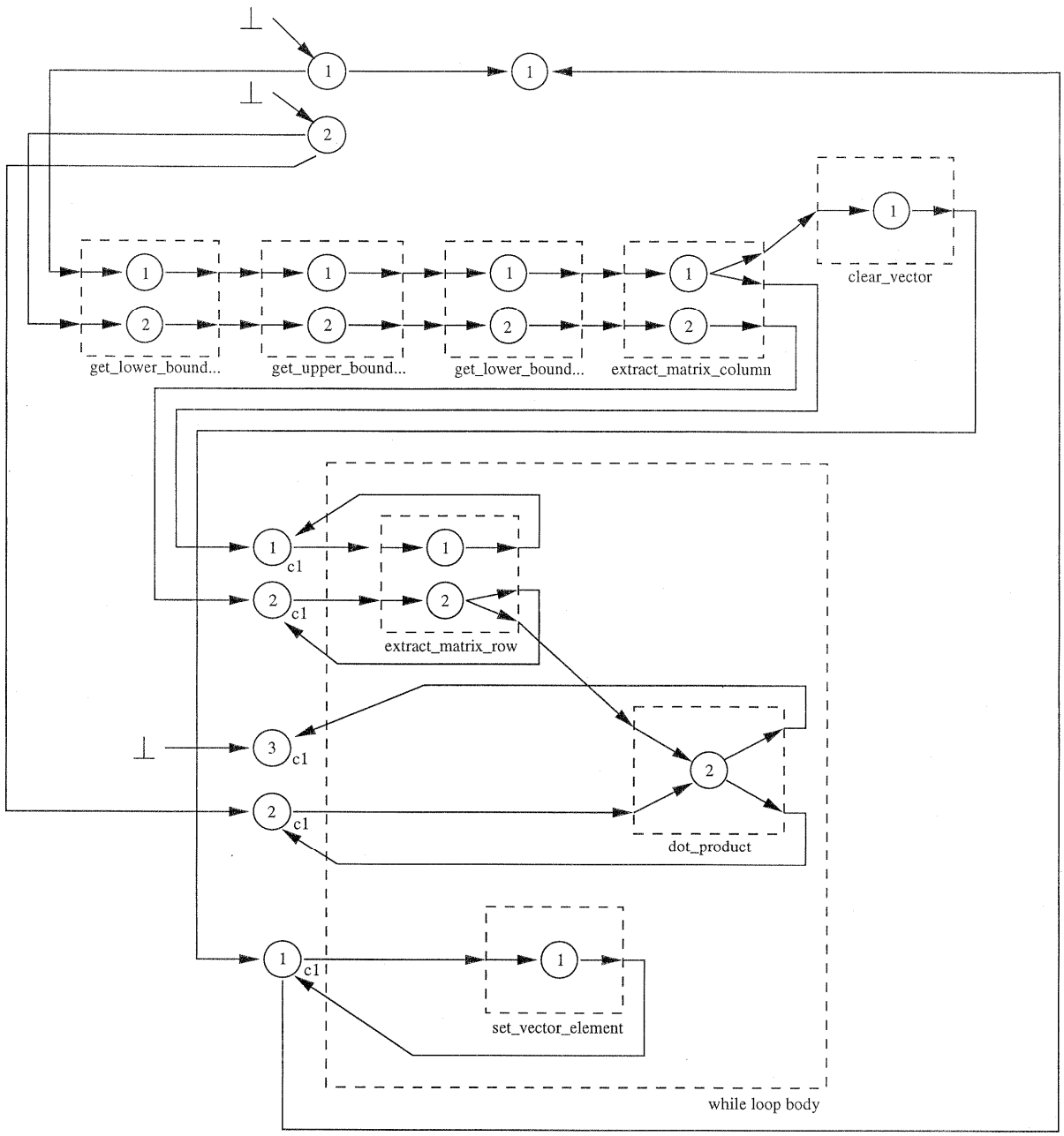
The justification for this definition of a legal abstract extent graph labeling is that it correctly captures a portion of the runtime behavior of programs. A brief sketch of the required proof is given in Appendix A.

7.3 Using the resulting graph to perform type checking

To make sure that the types are in fact correct, it suffices to check that all of the introduced vertices that corresponded to extents (either in procedure calls or in procedure definitions) are uniform. Once a solution to the abstract extent graph is computed, performing the actual type checking is therefore a very simple procedure.

8 Solving the Abstract Extent Graph

The only remaining obstacle to implementing an abstract extent-based sizing system is to construct a reasonable method for solving the generated abstract extent graph. In other words, we must find an algorithm for computing a function τ that associates with each vertex V of the abstract extent graph an abstract extent $\tau(V)$, using a minimal number of abstract extents subject to the conditions from Section 7.2.



Note: All unlabeled vertices have unique context labels.

Figure 6: Solution to the Abstract Extent Graph for the Example Program in Figure 2

We begin by noting that the conditions

$$\tau(x) = \tau(y) \Rightarrow (c(x) = c(y) \wedge \forall_i \tau(S_i(x)) = \tau(S_i(y)))$$

and

$$\tau(x) = \tau(\perp) \Rightarrow x = \perp$$

are equivalent to conditions for minimizing a finite state machine that both produces output from the states (rather than the edges) and has final states (the $c(x)$'s represent the outputs generated by each transition, the vertex $S_i(x)$ represents the next vertex when character i is received from state x, and $\perp$ is the only final state). Therefore, it seems natural to look for an extension of an algorithm for minimizing finite state machines.

A well-known algorithm for performing finite state machine minimization involves keeping at each stage of the process a collection of sets, such that two nodes that are in different sets are guaranteed to be non-equivalent [9]. The algorithm begins with two sets, one containing the final states, and one containing the non-final states. Sets which can be proven to be inconsistent are split into two or more pieces, until all of the sets are consistent. These sets then represent the states of the minimized machine. The algorithm runs in $O(v^2)$, where v is the number of vertices in the graph.

In a similar fashion, we can construct an nearly identical algorithm for solving abstract extent graphs. Here, the sets contain vertices that could possibly be associated with the same abstract extent. The algorithm proceeds just as for finite state machines, except that the set splitting must be done differently.

To attempt to split a given set of vertices S, we begin by temporarily setting aside those vertices that are uniform with respect to the current set assignments. These are the nodes whose ancestors are all members of the set S, and can be performed in $O(|S|)$ time. Having done so, for any two of the remaining nodes x and y to have the same abstract extent, we must have

$$(c(x) = c(y) \wedge \forall_i \tau(S_i(x)) = \tau(S_i(y)))$$

(since x and y are not uniform, nor will they *ever* be; the algorithm never recombines divided sets). We can therefore divide the non-uniform nodes into equivalence classes, where each equivalence class contains all the vertices with a particular context label and particular ancestor sets. This exactly parallels the equivalence class splitting done in a finite state machine simplification.

The only remaining detail is to re-integrate the uniform members of S that were separated initially. Intuitively, these vertices might be uniform in one of the new sets we have created, or they might have become non-uniform. Those vertices that are uniform in one of the new sets need to be added to that set. The remaining nodes (those which have just become non-uniform) are guaranteed to have different abstract extents than all of the existing equivalence classes, and so can be safely placed into their own separate set.

The algorithm described above can be implemented in time $O(|V|)$, where V is the equivalence class being split, if appropriate data structures are chosen (we omit the details for brevity).

We can then use this algorithm as the major subroutine of an algorithm for computing the solution to an abstract extent graph. We begin with two vertex sets: $\{\perp\}$ and $V - \{\perp\}$. This ensures that $\perp$ has a different label from all other nodes. We then begin attempting to split all existing equivalence classes, using the algorithm described above. The set of new equivalence classes generated is then split again. This process is iterated until no splittings are generated, at which point the algorithm is complete. Each vertex is then labeled with the equivalence class it is a member of, and the solution is complete.

8.1 Order of the Algorithm

If implemented carefully, our splitting algorithm can be implemented in $O(|S|)$ time, where S is the set to be split. Using this algorithm allows the complete solution to be computed in time $O(|V|^2)$.

As described in this paper, the construction of the abstract extent graph could produce a graph of size $O(vs)$, where v is the number of variables, and s is the size of a given procedure (number of lexical elements). In practice, sizes of order vs can only occur when the nesting depth of control structures is proportional to the size of the procedure. Since this is not typical of real programs, the size of the graph will be in practice considerably smaller, and would typically be closer to $O(s)$ than $O(vs)$.

The final phase is to use the information recorded in the abstract extent graph to verify the correct typing of the operations within the procedure. This involves checking that certain nodes of the AEG are uniform, and can be performed in $O(|V|)$. Therefore, the entire process can be solved in worst case time $O(v^2s^2)$.

Since algorithms exist which can minimize finite state machines in time $O(n \log n)$, where n is the number of vertices, it seems reasonable to look for an algorithm for solving a abstract extent graph with the same time bound. Unfortunately, our attempts to formulate such an algorithm (based around the techniques used in [8], for example), have not been successful to date. Further investigation into the possibilities for an $O(n \log n)$ algorithm for solving AEGs will be a important part of future research.

9 Future Work

Future work in this area of research may improve these results in a number of ways. Some obvious directions of further investigation include:

1. Looking at other $O(n \log n)$ algorithms for FSM minimization, to see if one of them will apply to solving AEGs.
2. Looking for a way to construct the abstract extent graph for a procedure that has size $O(s)$, where s is the size of the code, and not $O(sv)$, where v is the number of variables used, as the current algorithm does.

3. Examining whether it is feasible to remove the requirement for full procedure declarations, as do standard deductive type systems.
4. Considering the possibilities for merging sizing analysis into a standard deductive type framework, such as Hindly-Milner.

10 Summary

Languages for handling matrix-oriented computations would benefit from a typing system which could ensure that any requirements for size agreement between the arguments to a function are met at compile time. The advantages of this scheme are many, including:

1. The reduction of programming errors, which may not be detected until a set of inputs is found for which the errant code is actually executed,
2. Elimination of "error propagation", in which a low-level routine generates errors that were actually caused by a higher-level procedure being called with wrong argument sizes. These arguments tend to get passed down from level to level, producing incomprehensible error messages.
3. Improvements to efficiency, by making runtime checks of array sizes unnecessary in most cases.

In order to be useful, such a scheme must allow the user to specify the relationships between sizes, without specifying the exact sizes themselves. It must also be non-heuristic in nature, but rather have a formal, logical specification.

This paper has presented a system for implementing such size checking for a highly simplified language. It uses full declarations of formal parameter sizes to deduce the sizes of intermediate variables, and is then able to verify the correctness of the actual parameters. The algorithm currently runs in $O(s^2v^2)$, where s is the size of the procedure and v is the number of variables. We suspect that an algorithm of time $O(s \lg s)$ would be possible with further work.

We would like to acknowledge the contributions of Peter Lee and Paul Hilfinger, who have given us the benefit of their insights into our research.

A Correctness of Abstract Extent Graph

In this appendix, we outline a proof that solutions to the abstract extent of a procedure correspond to the runtime types of the original procedure. The main theorem required is to show that the abstract extent graph for a statement represents an abstract interpretation of the runtime type system:

> Theorem 1. Given a statement S has an abstract extent graph $G(I, O)$, where I and O represent the input and output vertices of the graph, respectively: If the variables on input to an execution of S satisfy the abstract extent associations $\tau(I)$, then after execution of S, the variables on output will satisfy $\tau(O)$.

By "satisfy $\tau(I)$", we mean that if $\tau(I(x,i))$ (the abstract extent assigned to the input vertex for dimension i of variable x) $= \tau(I(y,j))$, then the ith dimension of x has the same size as the jth dimension of y.

This theorem is proven using structural induction on S, based on the graph construction rules given in Section 7.1. In order to complete the proof, however, we will need to use the following lemma from Appendix B.

> Lemma C: Given two sets of vertices A and B, if a set of variable values satisfies $\tau(A)$, and $S_i(B) = A$ for some i, and $c(B) = z$ for all B, then the variable values will also satisfy $\tau(B)$.

This is a fairly immediate consequence of the following lemma, which actually contains the meat of the problem:

> Lemma B. $c(x) = c(y) \ \wedge \ S_i(x) = v \ \wedge \ S_i(y) = w \ \wedge \ \tau(x) = \tau(y) \Rightarrow \tau(v) = \tau(w)$, *for all graphs generated by the algorithm in Section 7.1*

Note that without the italicized restriction, lemma B is in general false. These lemmas (and an additional required precursor) are proved in Appendix B.

We omit most of the cases of the structural induction required to prove theorem 1, and sketch here only the most difficult, that for the `while` loop. Other control structures can be dealt with similarly.

Consider the statement `S1 = while (V) do S2`. Let the abstract extent graph of `S1` be $G_{S1}(I1, O1)$, and that of `S2` be $G_{S2}(I2, O2)$ (which is a sub-graph of G_{S1}), and let the nodes introduced by the construction of G_{S1} from G_{S2} be called P. If the program variables on input satisfy $\tau(I1)$, then they must satisfy $\tau(P)$, by lemma C and the construction of G_{S1}. But this means they must satisfy $\tau(I2)$, and by structural induction, the result of executing the loop body once will satisfy $\tau(O2)$. It then follows that these new values also satisfy $\tau(P)$ (by lemma C), and therefore $\tau(I2)$. This argument can be repeated any number of times, to show that all values generated by the while loop satisfy $\tau(P)$. If the while loop ever terminates, the resulting values must therefore satisfy $\tau(P)$. But the construction of G_{S1} implies that these values must also satisfy $\tau(O1)$, which completes the proof. Note that if the while loop does not terminate, then the program never produces any output variables at all (which trivially satisfies the theorem).

Given theorem 1, all that remains is to show that the initial values of variables are correctly represented by the part of the graph constructed by the procedure itself. This is very similar to the proofs already given, and we omit the details.

B Related Lemmas

In this appendix, we prove three results critical to proving the correctness of the abstract extent graph representation of our typing system (namely, that it represents an abstract interpretation of the runtime typing system).

Lemma A: For any vertex V in an abstract extent graph constructed according to the rules given in Section 7.1, there exists a path from $\perp$ to V, that does not travel through any vertex with context label $c(V)$.

This lemma can be proven by structural induction on the procedure body (the details are omitted here). The proof basically says that every control structure has an "entry point", which implies that there is a path from the beginning of procedure execution, where the size of all variables is derived from $\perp$. This lemma will be true for the appropriate graphs for any reasonable language, because there must be some sort of start to procedure execution.

Lemma B: $c(x) = c(y) \ \wedge \ S_i(x) = v \ \wedge \ S_i(y) = w \ \wedge \ \tau(x) = \tau(y) \Rightarrow \tau(v) = \tau(w)$ *for all graphs generated by the algorithm in Section 7.1*

A proof of Lemma B goes as follows. Both x and y are either uniform or not uniform. We prove the lemma by cases.

Case 1: $uniform(x)$ and $uniform(y)$

In this case, the definition of uniform tells us that

$$\tau(v) = \tau(S_i(x)) = \tau(x) = \tau(y) = \tau(S_i(y)) = \tau(w).$$

Case 2: $\neg uniform(x)$ and $\neg uniform(y)$

In this case, we must have $\forall i : S_i(x) = S_i(y)$. Therefore,

$$\tau(v) = \tau(S_i(x)) = \tau(S_i(y)) = \tau(u).$$

Case 3: $uniform(x)$ and $\neg uniform(y)$

This is the difficult case. We prove it as follows:

Applying lemma A, consider the path from x back to $\perp$ that avoids nodes labeled with $c(y)$ (which equals $c(x)$). We now show that each node on this path is both uniform, and has abstract extent equal to $\tau(x)$. Let p be a vertex on this path that satisfies $uniform(p) \ \wedge \ (\tau(p) = \tau(x))$ and let q be the vertex on the path such that $S_i(p) = q$ for some i. Then, since p is uniform, $\tau(q) = \tau(x) = \tau(y)$. But since $c(q) \neq = c(y)$, the definition of an abstract extent function τ says that

$$uniform(q) \ \vee \ uniform(y)$$

But since $\neg uniform(y)$, it must be the case that $uniform(q)$. Thus, q is uniform, and has abstract extent equal to $\tau(x)$. Thus, we have proven that

$$uniform(q) \ \wedge \ (\tau(q) = \tau(x))$$

Thus, by induction, all nodes on the path have equal abstract extent. This must include $\perp$, since it is a node on this path. But we know $\perp$ has a unique label (by definition of a correct abstract extent function τ). Therefore, x must equal $\perp$. But since $\tau(y) = \tau(x)$, y must also equal $\perp$. Therefore, $v = w$, and the theorem is valid.

Case 4: $\neg uniform(x)$ and $uniform(y)$

This case is proved symmetrically to case 3.

Thus, we have covered all four logical cases, and the lemma is proven.

Lemma C: Given two sets of vertices A and B, if a set of variable values satisfies $\tau(A)$, and $S_i(B) = A$ for some i, and $c(B) = z$ for all B, then the variable values will also satisfy $\tau(B)$.

Proof: This lemma is an immediate consequence of Lemma B. The variables will satisfy $\tau(B)$ if, when two vertices in B have the same abstract extent, their corresponding variable dimensions are the same. But Lemma B implies in this case that the same abstract extents in A will also be equal. Since we know that the values satisfy $\tau(A)$, this means that the corresponding sizes will be the same. Thus, the lemma is satisfied.

References

[1] ANSI. *American National Standard Programming Language Fortran ANSI X3.9-1978.*

[2] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, December 1994.

[3] Dept. of Defense. *Military Standard : Ada Programming Language*, 1981. MIL–STD–1815.

[4] Thomas Derby, Robert Schnabel, and Benjamin Zorn. EQ: Overview of a new language approach for prototyping scientific computation. In Pingali et al., editor, *Seventh Annual Workshop on Languages and Compilers for Parallel Computing*, pages 391–405. Springer-Verlag, 1994.

[5] Thomas Derby, Robert Schnabel, and Benjamin Zorn. Design ideas for prototyping scientific computations: the EQ language. *Journal of Scientific Programming*, 1995. Accepted to appear.

[6] Thomas Derby, Robert Schnabel, and Benjamin Zorn. The relationship between langauge paradigm and parallelism: the EQ prototyping langauge. In B. Szymanski and B. Sinharoy, editors, *Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 329–332, 1995.

[7] N. Halbwachs and P. Cousot. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Tenth ACM Annual Symposium on Principles of Programming Languages*, 1978.

[8] J. E. Hopcroft. An n log n algorithm for minimizing the states in a finite automaton. In Z. Kohavi, editor, *The Theory of Machines and Computations*, pages 189–196. Academic Press, New York, 1971.

[9] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*, pages 67–71. Addison–Westly, Reading, Massachusetts, 1979.

[10] Paul Hudak et al. Report on the programming language Haskell: A non-strict, purely functional language version 1.2. *SIGPLAN Notices*, 27(5):R–1–R–164, May 1992.

[11] Paul Hudak and Joseph Fasel. A gentle introduction to Haskell. *SIGPLAN Notices*, 27(5):T–1–T–53, May 1992.

[12] ISO. *Fortran 90 Standard ISO/IEC 1539: 1991(E).*

[13] M. Karr. Affine relationships among variables of a program. *Acta Informatica*, pages 133–151, 1976.

[14] Priyadarshan Kolte and Michael Wolfe. Elimination of redundant array subscript range checks. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 270–278, 1995.

[15] Math Works Inc. *MATLAB User's Guide*, 1992.