Domain-Specific Type Inference for Library Generation in a Telescoping Compiler

Cheryl McCosh
Rice University
Houston, TX 77005
chom@rice.edu

Arun Chauhan
Rice University
Houston, TX 77005
achauhan@rice.edu

Ken Kennedy
Rice University
Houston, TX 77005
ken@rice.edu

ABSTRACT
Telescoping languages is a strategy for allowing users to develop code in high-level, domain-specific languages and still achieve high performance. It uses extensive offline processing of the library defining the language. This process speculatively determines the possible uses of the library subroutines and generates variants specialized toward those uses. LibGen is a telescoping-language system for generating high-performance Fortran or C libraries with multiple specialized variants from a single version of MATLAB prototype code. LibGen uses variable types to guide specialization. Previously, we have shown that the generated code has comparable performance to hand-coded and optimized Fortran libraries and that specialization on type is important for achieving high performance.

In this paper, we describe the type inference system necessary for LibGen to speculate on the possible variants of library procedures and to generate code. We develop the concept of *type jump-functions*, which describe the transfer of type information through and across procedures. To compute these type jump-functions, we develop a static type-inference approach that uses a constraint-based formulation and a graph-theoretical algorithm shown to be efficient under conditions met in most practical cases.

1. INTRODUCTION
The productivity of the scientific computing community could be dramatically improved if it were possible for application developers to produce code by integrating components in high-level, domain-specific scripting languages, such as MATLAB. To this end, we have developed a framework, called telescoping-languages, that makes this possible by supporting high-level programming while achieving application performance comparable to code written in lower-level languages such as Fortran or C [22]. The framework accomplishes this by preprocessing the libraries that define the language to produce efficient variants specialized for specific calling contexts. The compiler for user-developed scripts can then replace calls to the general library subroutines with calls to the appropriate variants given the specific calling contexts. The strategy is shown pictorially in Figure 1.

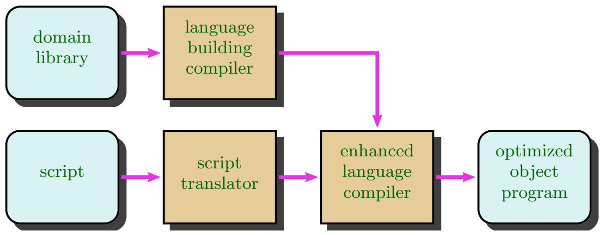

Figure 1: Overview of the telescoping languages approach.

In the process of our research, we have discovered that the telescoping-language strategy can be applied to the task of software library generation and maintenance. Library writers can develop and maintain a single version of their code in a high-level language. The compiler is then responsible for achieving high performance for the multiple possible uses of the library. It generates several Fortran or C variants specialized for the different possible calling contexts.

In previous work, we determined that type-based specialization was essential to achieve high performance, where the "type" of an array variable is extended to include intrinsic type, size, dimensionality and sparsity pattern [6]. It is therefore important to have numerous variants specialized to the different possible types of input parameters to the library routines. We showed that, if this is done, the library procedures automatically generated from MATLAB prototype code had runtimes comparable those of the hand-coded Fortran library. This paper elaborates the type inference framework used to obtain these results.

Because the library-building compiler, LibGen, must analyze the library procedures without knowledge of the calling context, during the type-inference phase, it generates *type jump-functions*, which summarize the types of the local variables in a procedure in terms of the types of the inputs. A variant for each possible configuration of input types is then generated. In order to avoid code explosion, the type inference system must determine which variants might actually occur in practice and lead to significant performance improvement.

Unfortunately, existing type-inference solutions for MATLAB do not meet these needs. We, therefore, developed a new approach that uses a propositional, constraint-based formulation and a graph-theoretical algorithm to find the solution. We show that the algorithm is efficient under reasonable assumptions and that it discovers all valid configurations of variable types. The algorithm works well with annotations from the library writer in that it can use the annotations to tighten constraints on the variables. The solution produced by the algorithm provides a basis for a dynamic phase of type inference called slice hoisting [5] and ultimately to the generation of specialized variants.

We begin with a brief discussion of the static type inference problem in MATLAB. We then introduce our implementation strategy and propositional formulation. This is followed by a description of our solution, first for single procedures, then for the interprocedural problem. In Section 5, we demonstrate how our solution can be applied to inferring MATLAB types. We show the results of our experiments in Section 6.

2. TYPE INFERENCE PROBLEM

Inferring types is important in realizing the goals of language generation in telescoping languages. Type inference is necessary for determining which operations are being called and which variants should be generated. In MATLAB, type inference is needed to translate to Fortran or C where types are stated explicitly.

For the purposes of this paper, we restrict ourselves to a core of the MATLAB language used in an overwhelming majority of scientific applications. This excludes dynamic evaluation of strings as code, the rather primitive object oriented features, and structured types (although, the last can be handled by a very simple extension of our ideas).

2.1 Features of MATLAB

MATLAB's simplicity makes it popular among programmers, as is evidenced by the large number of licenses.[1] However, some of the very features that make MATLAB a desirable language for programming make it difficult for the compiler to translate to lower-level languages. Some of these features include:

1. MATLAB is weakly-typed. This makes inferring intrinsic types necessary to generate code in lower-level languages, all of which require explicit typing.
2. Variables can change types in the middle of the program, including arrays growing in the middle of a loop. Inferring the maximum size of a loop would avoid reallocating the array at every iteration.
3. Operators are heavily overloaded. For example, the $*$ operation can refer to both matrix-matrix multiplication and matrix scaling depending on the operand types. Determining whether a variable is a scalar or an array is important both for determining which variant to call and in determining the types of the variables defined by the statement.
4. All variables are treated as arrays, including scalars, which are 1×1 arrays. Therefore, all variables have array properties that must be inferred.

[1]Mathworks quoted the number of licenses after 2001 to be 500,000

These features not only make type inference essential, they also make it difficult to statically determine the tightest types of the variables. Obviously, assuming every variable to be the most general possible type would be correct, but would not present enough optimization opportunities.

MATLAB's features require that the notion of type be extended to include other properties of variables beyond intrinsic type in order to translate to efficient lower-level code.

2.2 Type Problems

In order to carry out specialization based on variable types, we first define the variable properties that are of interest. In general, the properties should be such that they can be encoded in the target language and the compiler for that language can leverage the information for optimization. They should also reflect the power of the source language. Since the focus of this work is on numerical applications highly oriented towards array manipulations, we use the 4-tuple definition of a variable *type* used by deRose [12]. We define a type to be a tuple $\mathcal{T} = <\tau, \delta, \sigma, \pi>$ where,

- τ is the intrinsic type of the variable (e.g., integer, real, complex).
- δ is the upper bound on the number of dimensions for an array variable, also called the rank. A tighter bound can be reached when the type inference system determines that the variable has a size of 1 in one or more dimensions. δ will always be greater than 2.
- σ is a tuple showing the maximum size of an array variable in each possible dimension. $\sigma^A =< 1, 1 >$ means that A is a scalar.
- π is the "pattern" of an array variable (e.g., dense, triangular, symmetric, etc.).

This list can be extended as needed for other languages and other problems.

To avoid coincidental sharing of names among unrelated values (and hence types) of variables, we assume that the function has been converted to SSA form [10] so that each use of a variable refers to exactly one definition. Redefinition of a section of an array results in a new array. Type inference treats each SSA renamed variable as a distinct variable. A post-pass recombines names as much as possible to avoid copying.

In many cases, the library writer may have intended multiple interpretations of the same MATLAB code. The overloaded operators and weak typing are, in fact, one of the reasons why MATLAB is an easy language to program. The compiler must account for all the intended possibilities.

In MATLAB programs, the type of a variable depends on:

1. the operation that *defines* the variable and
2. the operations that use the variable, since operations impose type restrictions on their inputs.

The first causes *forward propagation* of variable properties along the control flow, while the second causes *backward propagation*. The type-inference system must infer types in both direction to have the tightest outcome.

2.3 Type Jump-Functions

The LibGen compiler has the extra burden of inferring types in the context of telescoping languages where the calling context is unknown during library compilation. The type-inference system must define the types in terms of the inputs so that during script-compilation time when the calling context is available, the correct types for the local variables are known. To handle this, we define *type jump-functions* akin to those used in interprocedural analysis [3, 16]. *Return type-jump-functions*, which define the types of the outputs in terms of the types of the inputs, handle the propagation of type information across procedures.

3. ALGORITHMIC OVERVIEW

We define a *type configuration* to be a set of valid type assignments, one for each variable. The type-inference algorithm must be able to infer all valid type configurations on a given lattice for a given subroutine. Each type configuration corresponds to a separate variant. Type configurations are tabular representations of type jump-functions discussed in the previous section.

3.1 Dataflow Frameworks

Telescoping languages proposes pre-compiling libraries before the calling-context is known. Therefore, MATLAB type-inference systems such as FALCON and MaJIC will not suffice, since FALCON relies on inlining to exactly determine types, and MaJIC determines types, in part, during a just-in-time compilation step. Moreover, both systems rely on dataflow analysis that converges on a single type for each variable.

There are two main difficulties to using a dataflow analysis framework.

1. It is difficult, if not impossible, to determine that the analysis halts on a given subroutine. This is partially due to the fact that information flows in both directions, but also, the lattices for some of the components of $\mathcal{T}$ do not meet the requirements for provable termination, specifically a finite-depth lattice.
2. The compiler must find all of the type solutions allowed by the function. Therefore, dataflow analysis is ill-suited for the problem, since if the analysis converges, it converges to a single solution (as in FALCON) or to a single general set of types. In the first case, there is not enough specialization opportunity. In the second case, the compiler generates more variants than necessary for calling contexts that would never occur. For example, one variable may prove to be complex only if the input is complex. Therefore, a variant would not be needed for the case where the input is real and the variable is complex.

Rather than solving a dataflow framework, we propose using a propositional formulation.

3.2 Propositional Formulation

We developed an alternative to the dataflow solution that analyzes the whole procedure simultaneously rather than iteratively. The compiler determines the information that each individual operation or procedure call gives about the types of the variables involved and then combines that information over the entire procedure. Thus, forward and backward inference occur simultaneously.

The information from the operations is given in the form of propositional *constraints* on the types of variables involved in the operation. MATLAB operations, and typical library procedures, are heavily overloaded. Therefore, a type constraint on an operation needs to allow all possible valid type configurations on the variables involved. The possible type configurations in a constraint are composed through logical disjunction and are called *clauses*. Figure 2 shows an example of size constraints on the MATLAB multiplication operation, "*". These clauses are defined to be mutually exclusive. Thus, each clause represents a distinct type configuration. The constraints are formed using a database of return type-jump-functions containing one entry per procedure or operation.

In a correct program, the type of a variable must satisfy all the constraints imposed by all the operations that can feasibly execute in any run of the program. This is equivalent to taking a conjunction of the constraints over the function and finding all possible type configurations of variables that satisfy the resulting boolean expression. This problem is NP-hard.[2] However, under certain conditions that occur most frequently in practice, we can devise an efficient algorithm using the specific properties of the problem.

4. AN EFFICIENT ALGORITHM

We start by describing the general algorithm for inferring types, and then discuss how to apply it to the type problems for MATLAB in the subsequent section.

First, there are a number of assumptions necessary for this algorithm to perform correctly.

1. The type-inference engine has valid code on input (i.e., all variables are defined before being used and the types of the variables are consistent).[3]

[2]The well known 3-SAT problem can be reduced to this problem.

[3]This is a reasonable assumption for MATLAB programs since users can develop and test their code in the MATLAB interpreter before giving them to the optimizing compiler.

```
c = mlfMtimes (a, b)
```

$$
\begin{array}{llll}
\sigma^{\mathbf{c}} = <1,\ 1> & \&\ \sigma^{\mathbf{a}} = <1,\ 1> & \&\ \sigma^{\mathbf{b}} = <1,\ 1> & | \\
\sigma^{\mathbf{c}} = <\$1,\ \$2> & \&\ \sigma^{\mathbf{a}} = <1,\ 1> & \&\ \sigma^{\mathbf{b}} = <\$1,\ \$2> & | \\
\sigma^{\mathbf{c}} = <\$1,\ \$2> & \&\ \sigma^{\mathbf{a}} = <\$1,\ \$2> & \&\ \sigma^{\mathbf{b}} = <1,\ 1> & | \\
\sigma^{\mathbf{c}} = <\$1,\ \$3> & \&\ \sigma^{\mathbf{a}} = <\$1,\ \$2> & \&\ \sigma^{\mathbf{b}} = <\$2,\ \$3> & | \\
\sigma^{\mathbf{c}} = <1,\ 1> & \&\ \sigma^{\mathbf{a}} = <1,\ \$1> & \&\ \sigma^{\mathbf{b}} = <\$1,\ 1> &
\end{array}
$$

Figure 2: Example of a constraint for the size inference problem on the MATLAB "*" operator which is internally implemented as a call to the library function called `mlfMtimes`. The \$-variables are simply place holders for integer values representing sizes. Each \$-variable is only used for a single constraint. This constraint keeps track of when a variable is a scalar or an array. If a \$-variable appears in the clause, it is assumed that the variable cannot be a scalar, although \$-variables may evaluate to 1. The first clause states that the operation is scalar multiplication. This second and third clauses state that this is a scaling operation. The fourth clause shows the operation is matrix-matrix or matrix-vector multiplication. The last clause gives the possibility that the operation is the multiplication of two vectors. This last case is necessary to keep track of the fact that c may be scalar. Otherwise, if c were used later, the compiler would assume it was not scalar and would make incorrect inferences based on that assumption.

2. All global variables have been converted to input and output parameters.

3. The number of input and output parameters in each operation or procedure is bounded by a constant. This property is important to limit the complexity. Parameter lists do not grow with the size of the procedure [8]. Also, few programmers use global variables excessively.

4. To form the operation constraints, the algorithm requires the compiler to already have return type-jump-functions on all the operations and procedures called by the procedure.

4.1 Reducing to the Clique Problem

After the constraints for each operation have been determined, the compiler must reason about them over the whole procedure. That is, it must find all possible type configurations that satisfy the whole-procedure constraint. By representing the operation constraints as nodes in a graph, the problem is reduced to finding n-cliques, where n is the number of operations in the procedure.

Figure 3 shows a simple example of how the graph is constructed for size constraints. Each possible type configuration for that operation, or clause, is represented by a node at the level corresponding to its statement. There is an edge from one node to another if the expressions in the nodes do not contradict one another. There is no edge from $1b$ to $2b$ since c cannot be both scalar and non-scalar. Note that since each clause is mutually exclusive, there is no edge between nodes on the same level.

The final graph has n levels, where n is the number of operations or procedure calls. Each level is bounded by l^v nodes,

```
A = b + c
```

$$
\begin{array}{lllll}
\mathbf{1a} & \sigma^{A} = <1,\ 1> & \&\ \sigma^{b} = <1,\ 1> & \&\ \sigma^{c} = <1,\ 1> & | \\
\mathbf{1b} & \sigma^{A} = <\$1,\ \$2> & \&\ \sigma^{b} = <1,\ 1> & \&\ \sigma^{c} = <\$1,\ \$2> & | \\
\mathbf{1c} & \sigma^{A} = <\$1,\ \$2> & \&\ \sigma^{b} = <\$1,\ \$2> & \&\ \sigma^{c} = <1,\ 1> & | \\
\mathbf{1d} & \sigma^{A} = <\$1,\ \$2> & \&\ \sigma^{b} = <\$1,\ \$2> & \&\ \sigma^{c} = <\$1,\ \$2> & |
\end{array}
$$

```
E = c - d
```

$$
\begin{array}{lllll}
\mathbf{2a} & \sigma^{E} = <1,\ 1> & \&\ \sigma^{c} = <1,\ 1> & \&\ \sigma^{d} = <1,\ 1> & | \\
\mathbf{2b} & \sigma^{E} = <\$3,\ \$4> & \&\ \sigma^{c} = <1,\ 1> & \&\ \sigma^{d} = <\$3,\ \$4> & | \\
\mathbf{2c} & \sigma^{E} = <\$3,\ \$4> & \&\ \sigma^{c} = <\$3,\ \$4> & \&\ \sigma^{d} = <3,\ 1> & | \\
\mathbf{2d} & \sigma^{E} = <\$3,\ \$4> & \&\ \sigma^{c} = <\$3,\ \$4> & \&\ \sigma^{d} = <\$3,\ \$4> & |
\end{array}
$$

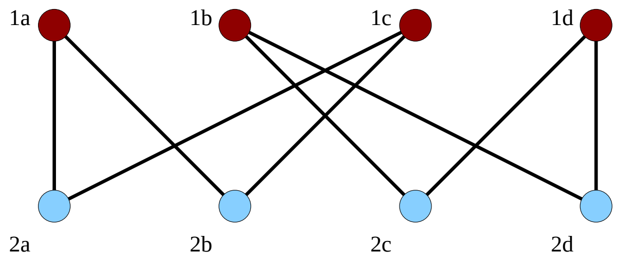

Figure 3: Example graph.

where l is the number of entries in the type lattice, which is assumed to be bounded by a small constant, and v is the number of variables involved in the operation, also assumed to be small by the third assumption. l^v is the number of possible type configurations for the variables in the operation corresponding to that level, since there are l possibilities for each variable. Since both l and v are assumed to be bounded by small constants, l^v is bounded by a constant.

Finding possible constraints over the entire procedure corresponds to finding sets of clauses that do not contradict each other such that there is one clause from each operation or procedure call. Having at least one clause from each operation constraint is necessary since otherwise, the resulting types will not hold over the entire procedure. On the graph, this is exactly the problem of finding all n-cliques (where n is the number of levels in the graph, and a clique is a complete subgraph) such that each n-clique has exactly one node from each level.

4.2 Description of the Algorithm

In order to show that using cliques to determine types is viable, we must show that the number of type configurations is bounded by a small number, and from this, that the total number of n-cliques is bounded by a small number.

Let *u-vars* be defined as the smallest set of variables such that all other variable types can be determined from the types of the u-vars. U-vars represent the set of variables that cannot be exactly determined statically (i.e., input types).[4] For the simple case where all operations are input-dependent, u is bounded above by the number of input parameters and is therefore small.

We define a *valid* procedure to be a procedure with the

[4]The algorithm may be able to infer exact types for some or all of the u-vars by their subsequent uses.

property that all definitions of variables occur lexically before any of their uses. In the case of control flow, ϕ nodes ensure this property for correct code.

THEOREM 4.1. *The number of possible type configurations is bounded by l^u, where l is the size of the type lattice and u is an upper bound on the number of u-vars.*

Proof: By the definition of u-vars, all other variable types excepting the u-vars can be statically determined in terms of the types of the u-vars. Therefore, the total number of possible configurations over all the variables is just the number of possible configurations of the u-var types. This is l^u since each u-var could potentially take on l types. $\square$

THEOREM 4.2. *The number of n-cliques is bounded by l^u.*

Proof (by contradiction): We start by assuming there are two distinct cliques that represent the same type assignment to the variables. The cliques must differ at least at one level. Since expressions in nodes of the same level contradict each other (hence, the need for the clauses to be mutually exclusive), at least one variable must have a different type. Therefore, the two cliques cannot have the same type assignment to the variables. $\square$

Finding n-cliques is $\mathcal{NP}$-Complete. However, we claim that given the structure of the problem, there is an algorithm that finds n-cliques in polynomial time given a bound on u.

The solution must be able to take advantage of the specific properties of the problem. Figure 4 gives a simple iterative algorithm to achieve this. The algorithm starts with one level and puts each node in that level in its own clique. For each subsequent step, it compares each node in the current level with each already formed clique. If the node has an edge to every member of the clique, it forms a new clique with the old clique. It does this until it reaches the last level. Figure 5 demonstrates the algorithm on an example graph. In each step, the lighter nodes and edges are part of one or more cliques. Because this algorithm follows the structure of the graph, it is easier to show that certain properties hold.

The loop starting on line 4 in Figure 4 iterates over the cliques from the previous step. Because the bound of l^u only holds for the final number of cliques, we need to find a bound on the number of intermediate cliques to limit the complexity.

THEOREM 4.3. *The number of cliques at each step of the iterative n-clique-finding algorithm is bounded by l^u if the levels are visited in program order.*

Proof: We have from above that on valid procedures there is an upper bound of l^u on the number of cliques. At any operation or procedure call, the rest of the code can be left off and the remaining (beginning) code is still valid. Therefore, since every iteration of the algorithm has processed valid code, if the levels are in program order, after every iteration, the algorithm will have produced cliques on valid code. The number of cliques after every iteration must be bounded by l^u. $\square$

```
input:  graph G
output:  CurrCliques
initialize CurrCliques to be nodes on first level
1 for every level r in G after first
2   newCliques = empty
3   for every node n in r
4     for every clique c in CurrCliques
5       candidate = true
6       for every node q in c
7         candidate = candidate & edge?(n,q)
8       end for
9       if (candidate)
10        then newCliques = newCliques + clique(c, n)
11    end for
12  end for
13  CurrCliques = newCliques
14 end for
```

Figure 4: Iterative n-clique finding algorithm.

With l^u cliques after every iteration, the n-clique-finding algorithm takes $l^v l^u n^2$ steps, where n is the number of operations, l^v is the maximum number of nodes in a level, and u is the number undeterminable variables. Therefore, the overall time complexity is $O(n^2)$ if l^u is bounded by a constant, which is true when all operations are input dependent.

Of course, in MATLAB, not all operations are input dependent. For some operations, the types of the outputs could depend on the values of the inputs rather than the types. This means that an operation could produce multiple output types on a given set of input types. Variables defined by such operations are u-vars.

Also, loops as well as branch statements use control-flow constructs. In the SSA representation ϕ-functions represent a merger of variable values under the assumption that every control-flow branch can be taken. As a result, ϕ-functions also represent the *meet* operation for types. The ϕ-nodes themselves do not need to be dealt with explicitly by the algorithm, but the introduction of a new variable defined by the ϕ-node causes an u-var if used later in the program.

Both of these situations increase u to the number of inputs and the number of variables defined by non-input-dependent operations or ϕ-nodes in pruned SSA form.[5] Although u could now be as large as the number of operations and procedures in the function, in actuality, this number should still remain small, since few operations are not input-dependent, and the amount of control flow usually remains small. Note that the algorithm works even without this assumption, but the complexity could become exponential in the worst case. If the amount of control flow does become large, the analy-

[5]We can get the benefit of working with pruned SSA without requiring the code to be in the pruned form, since if a variable is never used, it never appears in a constraint.

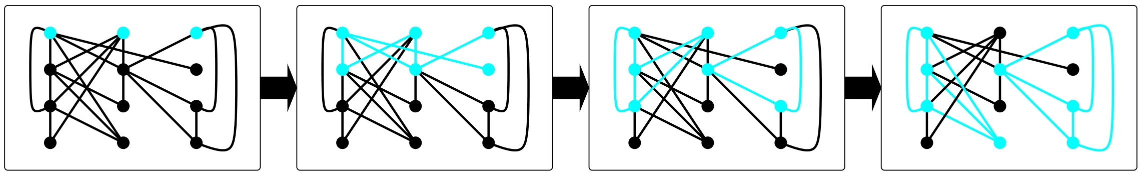

Figure 5: Example of n-clique-finding algorithm. Each row represents a level in the graph.

sis can split the graph at statements corresponding to join points, and the cliques can be merged when the all the pieces have been analyzed. This can greatly reduce the complexity.

4.3 Annotation Issues

One of the key ideas in telescoping languages is allowing the compiler to utilize the knowledge of the library writer through annotations. This information is important in knowing what cases can and cannot occur in practice, which the compiler alone may not be able to infer.

The library writer can provide type information on the parameters of the procedure to be analyzed. In the absence of source code, this will suffice for the return type jump-function. These annotations can also greatly help the compiler in analyzing the procedure. The compiler treats these annotations as constraints on the procedure header, which corresponds to the zeroth level in the graph. Any cliques occurring in the graph must have part of the user-defined annotations as one of their nodes. Annotations can greatly reduce the number of possible cliques and, therefore, specialized variants. They can also reduce the runtime of the algorithm.

4.4 The Result

The compiler needs a set of possible type configurations. Each clique represents a different type configuration. However, the equations in each clique still need to be solved to determine the type configurations. The compiler solves the equations using standard methods depending on the type problem.

For each resulting type configuration over the input parameters, the compiler creates an individual specialized and optimized variant. The appropriate variant is linked directly with the user script at script compilation time.

The compiler is not able to generate specialized variants for the added cliques corresponding to decisions from ϕ nodes and non-input-determined operations. However, the compiler can generate specialized paths from the control-flow points and operations if it can determine that optimizing would be beneficial. For example, the compiler would want to have separate paths if the outcome of the ϕ node could either be real or complex. Since this could cause an increase in the size of the code, the compiler must be careful about how many specialized paths are generated. Ultimately, the compiler could allocate the meet of the types to the variable on all paths unless the meet operation produces $\perp$.

In generating the optimized variants, the compiler is able to merge arrays treated as distinct by SSA if beneficial. The type of the resulting array is the meet of the types of all the variables being merged. This avoids wasteful copying.

Once the compiler has information about the inputs and outputs to the analyzed procedure, it computes a return type jump-function describing the library procedure. These will help in inferring types for calling procedures.

4.5 Interprocedural Inference

When the algorithm encounters a procedure call or a built-in operation, it looks in the database for the appropriate return type jump-function to build the constraints at that statement. Therefore, the algorithm assumes that these return type-jump-functions have already been computed for every called procedure. Therefore, type inference must have been performed in postorder on the call graph so that the children of a node are analyzed before the node itself. If the compiler encounters a procedure call for which there is no type jump-function, it simply considers the variables unconstrained by that statement. This degrades the analysis of the algorithm, although it does not affect the correctness. However, in the case of a cycle in the call graph or recursion, tighter information can be gained by iterating over the cycle until a fixed-point is reached. We have shown that a fixed-point can be reached in a constant number of iterations.

5. INFERRING $\mathcal{T}$

The algorithm described in the previous section can be applied to the type problems discussed in Section 2. The compiler infers each element of $\mathcal{T}$ in a separate pass and then takes the cross product of the different types to determine which variants are necessary. It can handle types separately since, except for δ and σ, the types are independent of each other, although knowing that a variable is scalar makes pattern inference unnecessary.

5.1 Inferring Number of Dimensions and Size

Before size inference can occur, an upper bound on the number of dimensions of an array is needed to determine the number of fields in σ. Only an upper bound is needed since the size inference will determine the actual dimensionality by inferring that some dimensions have size 1. Inferring dimensionality involves a single pass over the code to determine which dimensions of each variable are accessed explicitly or are used by an operation. If an upper bound cannot be determined (some operations have no limit on the number of dimensions), a dummy field in the size tuple is used to represent the sizes of dimensions beyond what is explicitly referenced.

Forming constraints for the size-inference problem has already been shown in Figure 2. Because of the infinite depth of the size lattice, the actual lattice used to determine compatibility consists of only information about whether the variable is a scalar or not. This means $l = 2$ for inferring sizes in the algorithm. Since many operators are overloaded based on these properties, they are necessary for the algorithm to make correct inferences. Since a variable cannot be both a scalar and a non-scalar, two nodes are not compatible if a variable is constrained to be a scalar in one node, and a non-scalar in the other.

The algorithm constrains the actual sizes of the variables by using \$-variables. Since each \$-variable can be used in multiple variable sizes, they capture the size relationships between the variables in a single operation. The fields in σ can be defined to be constants, \$-variables, or linear expressions involving both. Thus far, these expressions are sufficient to represent the size relationships between the variables.

Once cliques are found based on the scalar/non-scalar information, the actual sizes of the arrays in each dimension are inferred by solving the equations from the nodes in the clique using a variant of techniques used for solving linear Diophantine equations. All the variable sizes are computed in terms of the sizes of the u-vars (i.e., parameters, variables defined by ϕ nodes, etc.). The u-vars themselves may be statically inferred. In some cases, it is determined that a clique is invalid if there is no solution from the solver.

Through using a combination of two lattices, the algorithm is able to avoid the problems caused by the infinite-depth size-lattice. The dataflow framework is unable to track the size relationships over the finite lattice, and therefore cannot leverage benefit from such a split.

5.2 Inferring Intrinsic Type and Pattern

Intrinsic type and pattern inference differ only in the lattices used. The problems of inferring intrinsic types and patterns differ from inferring size in that the algorithm only operates on single, finite lattices. Therefore, the constraints are formulated differently. The constraints must restrict variables to a range of types on the respective lattice. Using ranges allows for values of types lower in the lattice than those defined by the parameter type specifications to be accepted as input. For example, an input argument that is defined as type `real` could actually be of type `int` when called. It also reduces the number of cliques, since each node in a level can represent multiple possibilities.

For the operation, $A = mean(x)$, the constraints are:
$(\texttt{real} \leq \tau^A \leq \texttt{real})\ \&(\perp \leq \tau^x \leq \texttt{real})|$
$(\texttt{comp} \leq \tau^A \leq \texttt{comp})\ \&(\texttt{comp} \leq \tau^x \leq \texttt{comp})$

Two constraint clauses are not compatible if a variable is in both clauses and its ranges do not intersect. The compiler still needs mutual exclusivity for the algorithm to run properly. Also, since maintaining the input dependence property is important in reducing the complexity, when possible, the constraints are formulated so that the same input configuration should give only one type of output.

Once the compiler has found the cliques, solving the equations corresponds to taking the intersection of all ranges for each variable over the clique.

The lattices for the intrinsic type and shape problems include a topmost element which is the most general case, a $\perp$, which represents invalid types, and intermediate elements. The meet between two elements is the top-most intersection of their paths from the bottom element.

LibGen allows the library writer to extend the type inference problem with new type problems not handled by size, shape and intrinsic type. The new problems, like intrinsic type and shape, must work on a single finite lattice. The library writer extends the inference problem by including a new base lattice for the new type problem.

6. EXPERIMENTAL EVALUATION

The algorithm discussed in this paper is currently being used as part of LibGen, a telescoping compiler for library generation. LibGen generates specialized Fortran or C variants based on the types inferred from the static analysis. The type configurations inferred by the algorithm from the MATLAB function ArnoldiC[6] are shown in Figure 6. The SSA form of ArnoldiC is used to show the relationship between the variables and their types in the type configurations. The type configurations shown can be exactly inferred given the types of the inputs. For example, if A_1 is non-scalar, then the last type-configuration should be used. This last configuration is, in fact, the only configuration intended by the library writers. An annotation stating that the input A is never scalar would have allowed the compiler to exactly infer the single type-configuration.

These three configurations are the valid subset of the automatically inferred configurations. In our current version of the compiler, we have not yet fully handled subscripted array-accesses, which is necessary to get the tightest constraints. We anticipate that this will be in place before the final submission deadline.

The actual number of inferred array types for both the size problem and the intrinsic type problem is shown in Figure 7 along with the compile times for inferring the configurations, including solving the cliques. The procedures used are all under forty lines. This is typical of the MATLAB development code, since programmers can describe their problems more succinctly in MATLAB.[7] All these procedures took under five seconds to compile despite the variations in code length and number of u-vars. The interprocedural solution is also implemented in the current version of the compiler. We show this with recursive code from a DSP library.

The compiler was run on a 1 GHz PowerPC G4 with the `-g` option.

7. RELATED WORK

MCC is the MATLAB to C compiler provided by Mathworks. The MCC output makes the library calls that would have been made by the interpreter. At the top level, the variables are still treated as arrays of a general type.

[6]This function is from the ARPACK development code.
[7]The hand-coded ARPACK version corresponding to ArnoldiC was over eighty pages long.

	config A	config B	config C
σ^{A_1}	$<1, 1>$	$<1, 1>$	$<\$1, \$1>$
σ^{v_1}	$<1, 1>$	$<\$1, 1>$	$<\$1, 1>$
σ^{k_1}	$<1, 1>$	$<1, 1>$	$<1, 1>$
σ^{v_2}	$<1, 1>$	$<\$1, 1>$	$<\$1, 1>$
σ^{w_1}	$<1, 1>$	$<\$1, 1>$	$<\$1, 1>$
σ^{α_1}	$<1, 1>$	$<1, 1>$	$<1, 1>$
σ^{f_1}	$<1, 1>$	$<\$1, 1>$	$<\$1, 1>$
σ^{c_1}	$<1, 1>$	$<1, 1>$	$<1, 1>$
σ^{f_2}	$<1, 1>$	$<\$1, 1>$	$<\$1, 1>$
σ^{α_2}	$<1, 1>$	$<1, 1>$	$<1, 1>$
σ^{V_1}	$<1,>$	$<\$1,>$	$<\$1,>$
σ^{f_3}	$<\$1, \$1>$	$<\$1, \$1>$	$<\$1, \$1>$
σ^{β_1}	$<1, 1>$	$<1, 1>$	$<1, 1>$
σ^{v_3}	$<\$1, 1>$	$<\$1, 1>$	$<\$1, 1>$
σ^{V_2}	$<\$1,>$	$<\$1,>$	$<\$1,>$
σ^{w_2}	$<\$1, 1>$	$<\$1, 1>$	$<\$1, 1>$
σ^{h_1}	$<j, 1>$	$<j, 1>$	$<j, 1>$
σ^{f_4}	$<\$1, 1>$	$<\$1, 1>$	$<\$1, 1>$
σ^{c_2}	$<j, 1>$	$<j, 1>$	$<j, 1>$
σ^{f_5}	$<\$1, 1>$	$<\$1, 1>$	$<\$1, 1>$
σ^{h_2}	$<j, 1>$	$<j, \$1>$	$<j, 1>$

```
function$[V, H, f] =$
    ArnoldiC$(A_1, k_1, v_1)$;
$v_2 = v_1/norm(v_1)$;
$w_1 = A_1 * v_2$;
$\alpha_1 = v_2' * w_1$;
$temp_1 = v_2 * \alpha_1$;
$f_1 = w_1 - temp_1$;
$c_1 = v_2' * f_1$;
$temp_2 = v_2 * c_1$;
$f_2 = f_1 - temp_2$;
$\alpha_2 = \alpha_1 + c_1$;
$V_1(:, 1) = v_2$;
$H_1(1, 1) = \alpha_2$;
for $j = 2 : k_1$,
    $f_3 = \phi(f_2, f_5)$;
    $\beta_1 = norm(f_3)$;
    $v_3 = f_3/\beta_1$;
    $H_2(j, j-1) = \beta_1$;
    $V_2(:, j) = v_3$;
    $w_2 = A_1 * v_3$;
    $h_1 = V_2(:, 1 : j)' * w_2$;
    $temp_3 = V_2(:, 1 : j) * h_1$;
    $f_4 = w_2 - temp_3$;
    $c_2 = V_2(:, 1 : j)' * f_4$;
    $temp_4 = V_2(:, 1 : j) * c_2$;
    $f_5 = f_4 - temp_4$;
    $h_2 = h_1 + c_2$;
    $H_3(1 : j, j) = h_2$;
 end
```

Figure 6: The resulting size configurations for all the variables and the corresponding pruned SSA form of ArnoldiC.

Name	*Compile Time*	*Lines*	*Size Cliques*	*Intr Cliques*
LUfac.m	4.29	15	72	27
ArnoldiC.m	4.29	25	32	18
QRcgsF.m	4.29	32	200	27
rec.m1	4.29	13	131	27
rec.m2	??	13	6	6

Figure 7: Type Inference time in LibGen for MATLAB Scripts

Type inference in FALCON and MaJIC from the University of Illinois at Urbana-Champaign is based on dataflow analysis [12, 11, 2, 1]. As noted earlier, this method is inadequate for our purposes.

MAGICA, a MATLAB type inference system developed at Northwestern, is able to infer array sizes in terms of compiler unknowns for multiple dimensions [21]. It uses algebraic systems of equations solved in Mathematica. We are not aware of any complexity results for this system. Also, MAGICA only infers a single set of explicit types that it can guarantee for all contexts. Otherwise, the array size is symbolic and left until runtime. This will not suffice for the telescoping compiler since it will not explicitly represent all possible configurations of scalars and non-scalars.

Recently, Elphick et. al. used partial evaluation of MATLAB code to help infer types [14]. For telescoping languages, partial evaluation will not suffice for the library compiler since it must guess about inputs before calling contexts are known.

The programming languages community uses a more general set of types in their type inference research. In particular, the issue of detailed arrays has been beyond the scope of their research. One exception is the work done by Xi and Pfenning [32] who use dependent types to check array bounds. The bounds of what their system will type check are more rigid. Several other areas have connections to our analysis of MATLAB.

Hindley-Milner type inference is closely related to our work [23]. Hindley-Milner type inference uses schema extensions for their principal types; we use type configurations.

The format of the lattices used for intrinsic type and pattern induces the notion of subtyping [26, 24, 4]. Subtyping with union and intersection types express similar properties to those needed in telescoping languages [9, 28, 27, 25]. Using intersection types has not yet been made practical for general purposes.

Techniques proposed in this paper can produce code to work with specialized libraries that are automatically tuned for specific platforms or problems such as ATLAS and FFTW [31, 15]. For example, specialized FFT routines can be called directly if the input matrix size is known.

Guyer and Lin have developed an annotation language for guiding optimizations on libraries [17].

Constraint Logic Programming (CLP) [29, 18, 19] extends the purely syntactic logic programming (typified by linear unification) by adding semantic constraints over specific domains. Some of the well-known CLP systems include CHIP [13], CLP($\mathcal{R}$) [20], Prolog-III [7], and ECL^{i}PS^e [30]. While a general purpose CLP system could be employed in solving the constraints within our type-inference system, our algorithm utilizes the properties of the problem to operate within a provably efficient time complexity.

8. CONCLUSIONS

We motivated the idea of speculative specialization of libraries based on types, which fits well within the telescoping languages framework. In order to carry out the specialization of code written in a weakly-typed, high-level language, like MATLAB, it is necessary to infer types of variables. Moreover, the inference process needs to generate all possible valid configurations of variable types based on the acceptable types of input parameters to a library procedure. Each such valid combination induces a specialized variant. In practice, the number of variants can be limited using annotations from the library writer.

We formulated the inference problem using propositional logic utilizing return type-jump-functions. The result of the formulation is a type jump-function that relates the types of the local variables to the possible input types. The type-inference system developed to do this is able to infer the smallest set of possible type-configurations.

Using the type-inference algorithm developed in this paper, we developed LibGen, a telescoping-language compiler that specializes and generates libraries based on type from MATLAB specification code. We have shown that the compiler can perform type inference in reasonable time and, in previous work, that the generated code is comparable to that of hand-coded Fortran libraries.

9. ACKNOWLEDGEMENTS

We thank Prof. Dan Sorensen for providing the motivation behind this work and example code as well as time and support. We would also like to thank Prof. Robert Cartwright and Prof. Walid Taha for their help with the related-work section.

10. REFERENCES

[1] G. Almási. *MaJIC: A Matlab Just-in-time Compiler.* PhD thesis, University of Illinois at Urbana-Champaign, 2001.

[2] G. Almási and D. Padua. MaJIC: Compiling MATLAB for speed and responsiveness. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 294–303, June 2002.

[3] D. Callahan, K. D. Cooper, K. Kennedy, and L. Torczon. Interprocedural constant propagation. *SIGPLAN Notices*, 21(7):162–175, July 1986.

[4] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2/3):138–164, Feb./Mar. 1988.

[5] A. Chauhan and K. Kennedy. Slice-hoisting for array-size inference in MATLAB. In *Languages and Compilers for Parallel Computing*, 2003.

[6] A. Chauhan, C. McCosh, K. Kennedy, and R. Hanson. Automatic type-driven library generation for telescoping languages. In *ACM/IEEE SC2003 Conference on High Performance Networking and Computing (Supercomputing)*, Nov. 2003.

[7] A. Colmerauer. An introduction to Prolog-III. *Commun. ACM*, 33(7):69–90, July 1990.

[8] K. D. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 57–66, Atlanta, GA, June 1988.

[9] M. Coppo, M. Dezani-Ciancaglini, B. Venneri, c of, and t Zeitschrift. ur mathematische logik und grundlagen der mathematik, 1981.

[10] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct. 1991.

[11] L. DeRose and D. Padua. Techniques for the translation of MATLAB programs into Fortran 90. *ACM Transactions on Programming Languages and Systems*, 21(2):286–323, Mar. 1999.

[12] L. A. DeRose. *Compiler Techniques for Matlab Programs.* PhD thesis, University of Illinois at Urbana-Champaign, 1996.

[13] M. Dincbas, P. V. Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems FGCS-88*, pages 693–702, Dec. 1988.

[14] D. Elphick, M. Leuschel, and S. Cox. Partial evaluation of MATLAB. In *Generative Programming and Component Engineering (GPCE'03)*, Lecture Notes in Computer Science, pages 344–363, 2003.

[15] M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 3, pages 1381–1384, May 1998.

[16] D. Grove and L. Torczon. Interprocedural constant propagation: A study of jump function implementations. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 90–99, June 1993.

[17] S. Guyer and C. Lin. An annotation language for optimizing software libraries. In *Proceedings of the Second Conference on Domain-Specific Languages*, Mar. 1999.

[18] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 111–119, 1987.

[19] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[20] J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. The CLP($\mathcal{R}$) language and system. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, July 1992. Also available as Technical Report, IBM Research Division, RC 16292 (#72336), 1990.

[21] P. Joisha and P. Banerjee. Implementing an array shape inference system for MATLAB. Technical Report CPDC-TR-2002-10-003, Center for Parallel and Distributed Computing, Department of Electrical and Computer Engineering, Northwestern University, Oct. 2002.

[22] K. Kennedy, B. Broom, K. Cooper, J. Dongarra, R. Fowler, D. Gannon, L. Johnson, J. Mellor-Crummey, and L. Torczon. Telescoping Languages: A strategy for automatic generation of scientific problem-solving systems from annotated libraries. *Journal of Parallel and Distributed Computing*, 61(12):1803–1826, Dec. 2001.

[23] R. Milner. A theory of type polymorphism in programming languages. *Journal of Computer and System Sciences*, 17(2):348–375, 1978.

[24] J. Mitchell. Type inference with simple types. *Journal of Functional Programming*, pages 245–285, 1991.

[25] B. C. Pierce. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, 1991.

[26] J. Reynolds. Using category theory to design implicit conversions and generic operators. In *Semantics-Directed Compiler Generation*, Springer-Verlag Lecture Notes in Computer Science, pages 211–258, 1980.

[27] J. Reynolds. Design of the programming language forsythe. Technical Report CMU-CS96 -146, Carnegie Mellon University, June 1996.

[28] P. Salle. Une extension de la theorie des types en -calcul. In *Springer-Verlag*, volume 62 of *Lecture Notes in Computer Science*, pages 398–410, 1982.

[29] G. L. Steele. *The Definition and Implementation of a Computer Programming Language based on Constraints*. PhD thesis, M.I.T., 1980. AI-TR 595.

[30] M. Wallace, S. Novello, and J. Schimpf. *ECL^iPS^e: A Platform for Constraint Logic Programming*. William Penney Laboratory, Imperial College, London, 1997.

[31] R. C. Whaley and J. J. Dongarra. Automatically Tuned Linear Algebra Software. In *Proceedings of SC: High Performance Networking and Computing Conference*, Nov. 1998.

[32] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 249–257, June 1998.