Automatic Type-Driven Library Generation for Telescoping Languages

Arun Chauhan achauhan@cs.rice.edu
Cheryl McCosh chom@cs.rice.edu
Ken Kennedy ken@cs.rice.edu
Richard Hanson koolhans@cs.rice.edu

Department of Computer Science, Rice University, Houston, TX 77005

ABSTRACT

Telescoping languages is a strategy to automatically generate highly-optimized domain-specific libraries. The key idea is to create specialized variants of library procedures through extensive offline processing. This paper describes a telescoping system, called ARGen, which generates high-performance Fortran or C libraries from prototype Matlab code for the linear algebra library, ARPACK. ARGen uses variable types to guide procedure specializations on possible calling contexts.

ARGen needs to infer Matlab types in order to speculate on the possible variants of library procedures, as well as to generate code. This paper shows that our type-inference system is powerful enough to generate all the variants needed for ARPACK automatically from the Matlab development code. The ideas demonstrated here provide a basis for building a more general telescoping system for Matlab.

1. INTRODUCTION

For several years, we and our colleagues at Rice have been conducting research and development on a strategy, called *telescoping languages*, for automatically generating high-level domain languages [12]. The central idea is to build these languages from a powerful scripting language, such as Matlab or Python, augmented with annotated libraries of domain-specific components. To make this strategy practical, we must compile programs to high-performance object code, so that developers do not need to recode them in a lower-level language, such as Fortran or C, before putting them into production use.

A tenet of the telescoping-language approach is that compile time for the scripts must be kept low if the language is to be usable—extensive interprocedural compilation of the script, plus all library routines invoked directly or indirectly from it, would take too much time to be accepted by the users. Hence, we have developed the new compilation approach depicted in Figure 1, in which speculative interprocedural optimizations on the library are performed in the language-building compiler, which is invoked only occasionally at "language-generation time."

SC'03, November 15-21, 2003, Phoenix, Arizona, USA

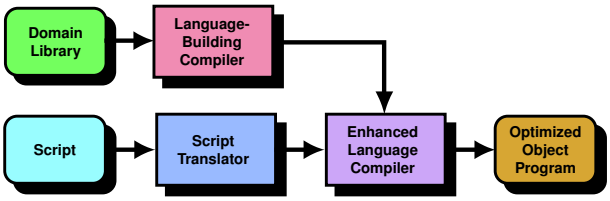


Figure 1: Graphical Model of Telescoping-Language Approach

In addition, the language-building compiler will read annotations provided by the library developer, usually concerning likely calling contexts for various routines and advice on how to replace general routines with more specialized versions tailored to those calling contexts. From those annotations, it will generate optimizations that are included in the enhanced language compiler, which is used to compile each script. These optimizations carry out the substitutions of context-specialized sequences of operations for the generalized versions invoked in the script. Thus, the enhanced language compiler optimizes invocations of the domain library routines as if they were primitive operations in the base language.

To date, we have explored the telescoping-languages strategy by generating prototype optimizations for Matlab [4] with signal processing and for a statistical language called S.

In the course of our research on telescoping languages, we discovered that our colleague Prof Dan Sorensen from the Dept. of Computational and Applied Mathematics (CAAM) used Matlab as a prototyping language for his linear algebra library, ARPACK. ARPACK, which stands for ARnoldi PACKage [15], is a collection of Fortran 77 subroutines designed to solve large-scale eigenvalue problems. ARPACK implements a variant of the Arnoldi process called *implicitly restarted Arnoldi method (IRAM)*.

Once the prototype was completed, Sorensen and his student painstakingly translated it into eight different Fortran variants, specialized to matrix class (symmetric versus unsymmetric) and data type (complex versus real, single versus

double precision), to achieve high performance. The amount of work involved in this translation—a half-page Matlab prototype translated to 12 pages of Fortran for each variant—prevented them from producing other desirable type-based specializations. In particular, it would be extremely useful to support generation of ARPACK for different sparse-matrix representations, a capability that is provided by a coroutine ("reverse communication") interface in the Fortran version.

This discovery led us to consider a different application of telescoping-languages technology, namely library specification, generation, and maintenance. If the library developer could specify, as *annotations*, all the desired type variants of input parameters to the library, then the enhanced language compiler might be able to automatically generate all of the desired variants in Fortran or C without any further intervention by the developer. If this approach could be made to work, it would yield enormous productivity gains for library developers.

To explore this idea, we have undertaken the development of *ARGen*—a telescoping library generator for ARPACK driven by type-based procedure specialization. In this system, the developer provides a Matlab prototype plus annotations specifying a variety of anticipated calling contexts, in terms of the types (matrix class and data type) of input parameters in each context. The ARGen system then generates a specialized variant for each context in a low-level language (Fortran in this case).

The key technical challenge is, given the types of input variables to a library prototype, to infer type information for Matlab variables at each point in the prototype. To solve this problem, we use a novel strategy that constructs, at each point in the Matlab routine, a *type jump-function*, which can be evaluated to produce types of local variables, once the actual types of the input variables are known. The construction involves a combination of static and dynamic analysis. The paper demonstrates the power of this approach by experiments showing the relative performance of Matlab and Fortran versions of ARPACK with the versions generated by ARGen.

We emphasize that the idea underlying the ARGen system is completely general, in that it could be used to generate variants for *any* library specified in Matlab. ARGen is simply the first demonstration, one that already has a customer in Prof. Sorensen.

2. IMPLEMENTATION STRATEGY

Through working on the problem of automatically generating the Fortran ARPACK from the Matlab code, it became clear that inferring types would be an important starting point in realizing the goals of telescoping languages. Since Matlab is weakly typed, type inference is necessary to translate to a lower-level language where types are stated explicitly. Type inference is also necessary for code generation, since the operators in Matlab are overloaded depending on the types of the input. Therefore, without knowing the types of the input, it would be difficult to determine the meaning of the operation.

Furthermore, accurate type inference is essential in determining which variants are beneficial to generate. While generating a single variant with the most general type would be correct without the presence of operators overloaded on input types, to produce the fastest possible optimized code, the tightest information on the types in the program is needed. Since often Matlab programmers intend for their code to be used for multiple types of input, it is important for the compiler to infer that these different types can occur. A separate variant will be generated if one of the possible type configurations would allow for substantial speedups over the others (with or without further optimization) or if the meaning of the program changes depending on types. Therefore, the variants will be generated so that each is not only possible, but beneficial from an optimization standpoint or necessary from a correctness standpoint.

Because telescoping languages proposes pre-compiling the library before the calling context is known, type inference systems for Matlab such as FALCON and MaJIC will not suffice, since FALCON relies on inlining to exactly determine types and MaJIC performs just-in-time compilation. Also, to infer types, both systems rely on data-flow analysis that converges on a single type for each distinct variable. Therefore, all possible type configurations are not inferred, which would lead to a limited number of variants with limited optimization. While data-flow analysis could determine a set of possible types for each variable if formulated correctly, it could not easily describe the relationships between the types of the variables. Tightening the relationship between variables would help to determine exactly which variants are necessary. For example, one variable may only prove to be complex if the input is complex. Therefore, a variant would not need to be generated for the case where the input is real and the variable is complex.

2.1 Features of Matlab

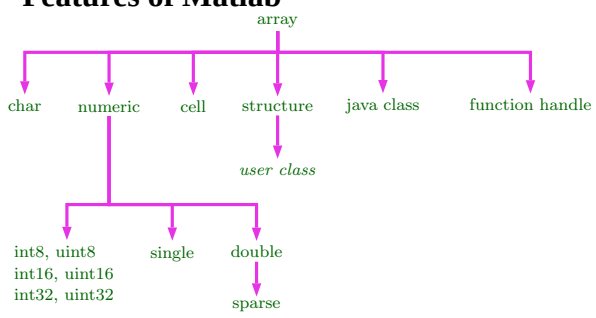


Figure 2: Matlab Type-Hierarchy

The simplicity of the Matlab syntax is well suited towards increasing coding productivity. However, some of the very features that make Matlab ideal for development purposes are a hindrance when translating applications to a lower-level language such as FORTRAN or C. These features include:

- Matlab is weakly typed. This means that inferring types is necessary to translate Matlab code into a lower-level languages, all of which require declared types.

- Matlab allows types to change if they are redefined in the middle of the subroutine. For example, arrays can grow, and each growth would require reallocating the array. This severely hampers performance when an array grows within a loop (which happens in ArnoldiC).

- As can be seen from Figure 2, Matlab treats all variables first and foremost as arrays, including scalars, which are represented as 1×1 arrays.

- Operations in Matlab are heavily overloaded. For example, in the statement `w=A*v`, the `*` operation could be interpreted as matrix multiply if both `A` and `v` are matrices. However, if `A` is a scalar, then the operation should be interpreted as multiplying every element of `v` by `A`. The operation used affects the size of w, which, in turn, affects other sizes that are defined using w, as well as the meaning of the statement.

Because of these features, the compiler often cannot statically determine an exact type for every variable. However, in some cases the library writer may have intended multiple interpretations of the same code. The overloaded operators are, in fact, one reason why Matlab is a much simpler language for developing code, since one operation or subroutine can handle many different kinds of input. The compiler must account for each of the intended possibilities while limiting the number as much as possible to avoid generating more specialized variants than are necessary.

ArnoldiC, a Matlab ARPACK subroutine, was intended for both complex and real arrays. In converting ArnoldiC into Fortran, ARGen needs to generate code for each possibility, not only because Fortran requires a variable's type to be declared, but also because for correctness and efficiency, different operations must be used depending on whether the matrix is complex or real. The compiler has to provide separate code for each case as well as code that determines types at runtime that could not be determined statically.

2.2 Definition of Type

The kind of type information needed by the compiler depends both on what the scripting language (Matlab) allows and also what the destination language (Fortran) needs. Because Matlab treats all variables as arrays, the compiler needs to use an extended notion of type (based on De Rose's work) [7, 8].

A variable's type, for the purposes of this paper, is defined as a tuple $<\tau, \delta, \sigma, \pi>$ where,

- τ refers to intrinsic type such as int, real, complex or char (needed by the compiler to declare a variable in Fortran),

- δ refers to an upper bound on the number of dimensions of the variable or dimensionality (needed to allocate space in Fortran),

- σ is a tuple showing the size of the array in each of the ρ possible dimensions (also needed for memory allocation in Fortran), and

- π is the pattern of the array, such as sparse, banded, etc. (useful for optimization).

Knowledge of these types is important for both code generation and specialization/optimization.

2.3 Need For Forward and Backward Flow of Information

Information about the type of a variable can be determined at the definition point based on the operation and the inputs involved. This type information can then be used at a later point when another variable is defined using the previously defined variable.

If the variable's exact type could not be determined at its definition, information about its type can also be determined by the variable's uses. As one example, if the variable is an input to an operation that accepts only a known type, then it can be assumed, given correct code, that the variable is of that type. In Matlab, information about variable types flows both forward and backward.

Figure 3 shows a piece of code from the Matlab ArnoldiC subroutine, which demonstrates an example of where backward flow of information is useful. The size of `v` is not clear from the definition, but line 7 explicitly refers to `v` as a vector if `V` has size greater than one in that dimension. This information can lead to better information about variables defined by and used with `v`. By taking into account information flowing in both directions, tighter analysis is performed.

```
1   v = v/norm(v);
2   w = A*v;
3   f = w - v*alpha;
4   c = v'*f;
5   f = f - v*c;
6   alpha = alpha + c;
7   V(:,1) = v;
8   H(1,1) = alpha;
```

Figure 3: Example Where Backward Propagation is Useful

2.4 Whole-Program Analysis

There are two difficulties to using data-flow analysis for solving the type-inference problem:

1. It is difficult, if not impossible, to determine if the analysis halts on a given subroutine. This is due, in part, to the fact that information flows in both directions. Also, the lattices involved in solving the problem do not meet all the requirements for proving termination, namely a finite lattice and a monotonic meet operation (i.e. the result of the meet operation could be above or below its arguments on the lattice).

2. The compiler needs to find all of the solutions allowed by the problem. Therefore, data-flow-analysis is ill-suited for the problem, since if the analysis converges, it converges to a single solution or to a single general

1. parse the Matlab procedure
2. build constraints using the annotation database
3. use the n-clique algorithm to statically infer types
4. transform the code for dynamic size inference using slice hoisting, wherever applicable
5. generate C / Fortran code for specialized variants

Figure 4: High-level steps for ARGen.

set of types. In other words, it would not be able to compute the relationships between the possible variable types. In order to solve the problem using conventional data-flow analysis, the compiler would have to run several passes of analysis under all possible assumptions about the variable types.

Instead of data-flow analysis, we use a strategy developed by McCosh[13] in which type jump-functions that describe the possible argument types on each operation are transformed into propositional constraints on the statements. The constraints are then solved over the whole procedure using an graph-based, theoretical algorithm, which is efficient ($O(n^2)$) under reasonable assumptions about the code. The sizes that are not determined statically are found dynamically using a strategy called slice-hoisting developed by Chauhan[3]. This strategy uses dependence information to bring the allocation of arrays to the earliest possible point in the procedure, alleviating the need for dynamically reallocating arrays. Annotations provided by the library writer can make analysis tighter and more efficient, but are not necessary for the compiler to generate correct variants. This paper demonstrates the power of these strategies within the ARGen system experimentally.

3. EXPERIMENTAL EVALUATION

Figure 4 outlines the steps needed to specialize a Matlab procedure based on the types of its input and output values.

We have implemented a telescoping compiler for Matlab that specializes the libraries based on inferred types and library-writer annotations. The output code can be emitted in either Fortran or C. This section presents experimental results for a specific routine in ARPACK, called ArnoldiC. ARGen automatically generates code specialized on types from the Matlab prototype code that was used in designing ARPACK. ArnoldiC is a typical subroutine from the Matlab code. All other routines have very similar properties.

ARGen automatically infers intrinsic type, pattern, and size information and generates all valid configurations of types for the variables. A total of 12 size configurations are generated. In fact, only one of the inferred size configurations is what the library writers intended, but three size configurations are valid. The extra configurations are inferred due to an imprecision in handling indexed array references. The code generator automatically generates code in C or Fortran. The inferred type information is then used to specialize the generated code for each type configuration. The codegen phase uses a specialization table, that can easily be extended or modified, to drive this specialization process.

	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>

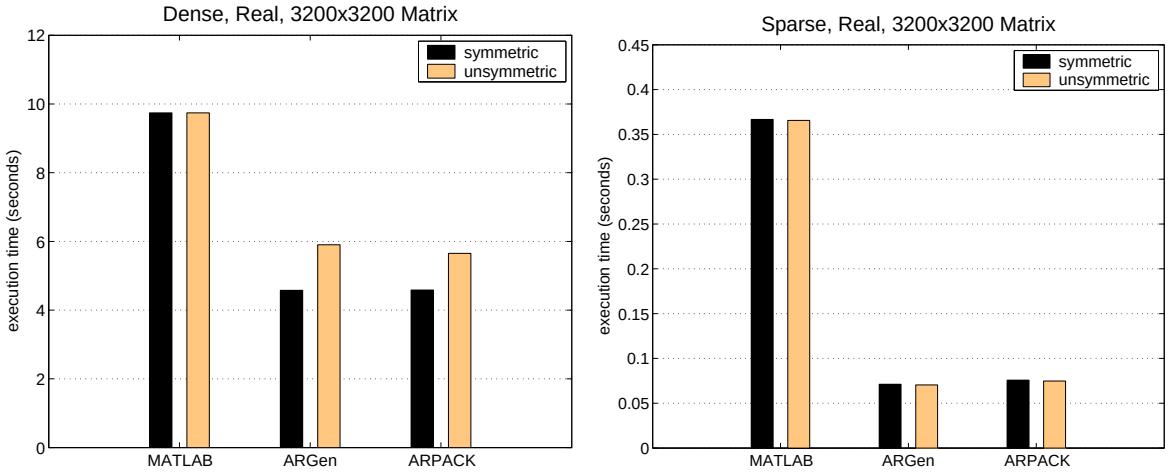
```
function[V, H, f] =
    ArnoldiC(A_1, k_1, v_1);
v_2 = v_1/norm(v_1);
w_1 = A_1 * v_2;
α_1 = v_2' * w_1;
temp_1 = v_2 * α_1;
f_1 = w_1 − temp_1;
c_1 = v_2' * f_1;
temp_2 = v_2 * c_1;
f_2 = f_1 − temp_2;
α_2 = α_1 + c_1;
V_1(:, 1) = v_2;
H_1(1, 1) = α_2;
for j = 2 : k_1,
    f_3 = φ(f_2, f_5);
    β_1 = norm(f_3);
    v_3 = f_3/β_1;
    H_2(j, j − 1) = β_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 5: The resulting size configurations for all the variables and the corresponding pruned SSA form of ArnoldiC.

Figure 5 shows the SSA form (see [5]) of the Matlab ArnoldiC—a procedure from the Matlab code—and the configurations resulting from performing the static size inference on it. Columns two through four list the sizes corresponding to the different configurations discovered by the size inference algorithm. This is the result when no annotation is supplied by the library writer on the input parameters. In fact, only the third configuration (config C) was intended by the library writers since `A` is always expected to be a matrix, never a scalar. An annotation on `A` stating this fact could automatically prune the configurations.

The Fortran code emitted by ARGen uses ATLAS-tuned BLAS for matrix operations. By using BLAS, the generated code is not only specialized for different possible types, but also for different architectures. Both the Fortran ARPACK and the Matlab interpreter use ATLAS-tuned BLAS, so the figures accurately reflect the speedups do to the type-based specialization. Further, for the purposes of this study, we also hand-specialized the output code for symmetric matrices to compare its effect with the standard non-symmetric version. This step will eventually be performed automatically. All experiments were conducted on SGI Origin and the Fortran code was compiled with `-O3` optimization option.

The hand-coded Fortran ARPACK suite allows an application to obtain accurate information about eigenvalues of large matrices. The programming required of the user is



(a) Dense matrix.

(b) Sparse matrix.

Figure 6: Comparing running times on MATLAB with ARGen and ARPACK.

code that gives the package the results of any requested matrix-vector product. This operation can be explicit—matrix times vector—or result from a linear operator or operation such as a sparse matrix-vector product or solution of a linear system. The ARPACK software iteratively returns to the calling program unit and requests the operation. Then the software is re-entered until the requested accuracy is obtained. This is called a *reverse communication* interface. ARGen does not emit code for reverse communication, but instead, uses the straight subroutine call to perform the matrix-vector multiplication.

3.1 Performance of ARGen

Figure 6 compares the execution time of ArnoldiC under MATLAB 6.5 with that generated by ARGen and the corresponding routine in ARPACK. The first sub-figure shows the running times for a dense matrix input of 3200×3200 elements for both symmetric and non-symmetric types, and the second sub-figure plots similar results for a sparse matrix that has around 18,000 non-zero values.

The performance of ARGen-generated code comes very close to the hand-coded ARPACK. The Fortran ARPACK suite applies steps to assure that the output matrix V is orthogonal. It is an additional computation that is not always needed. This DGKS correction is *always* applied in the ArnoldiC Matlab code, resulting in slightly more computation in the ARGen-generated code in some cases.

On the other hand, because of the reverse communication interface, the Fortran ARPACK library incurs extra overheads. The Matlab version (and hence, ARGen emitted code) does not have these overheads. However, we expect that such efficiencies will be relatively small compared to the times for computation of the matrix-vector products or linear operators of large-scale applications.

3.2 Value of Specialization

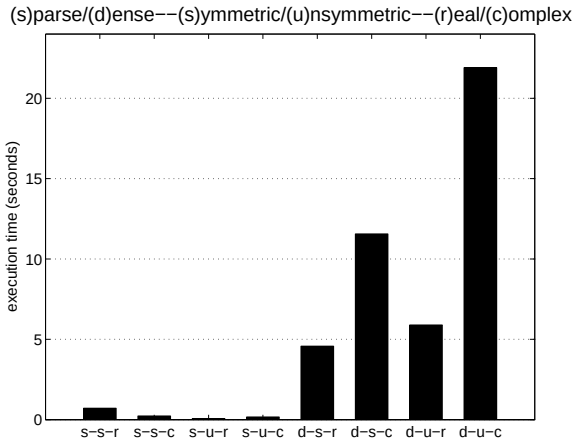


Figure 7: Value of Specialization.

Figure 7 shows running times of different ARGen generated specialized variants for the same input that is sparse and symmetric with real values. The chart clearly indicates that there is value in using the right type of specialized variant. An inaccurately inferred type or incorrectly used variant can potentially result in a huge performance loss.

The specialized variant that works on sparse, non-symmetric, real values runs slightly faster than that operating on sparse, symmetric, real values. We believe that this could be due to the inefficiencies of the sparse representation.

3.3 Effect of Scaling

We measured running times for different matrix sizes to assess the effects of scaling on type-based specialization. The results are shown in figure 8. As the input size grows, the

performance benefit scales almost perfectly. Thus, at larger input sizes the performance gain in absolute terms is even bigger.

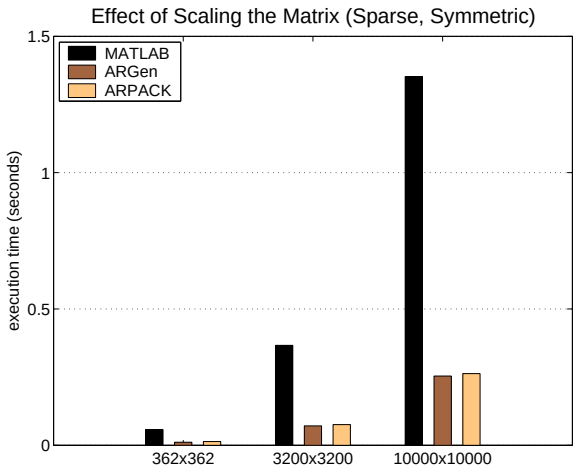


Figure 8: Performance of ARGen-generated code scales almost perfectly the input size.

4. RELATED WORK

The FALCON compiler at the University of Illinois carried out type inference for Matlab for generating code in Fortran 90 [8, 7]. The 4-tuple definition of types used in this paper has been motivated from that work. However, the type inference in the FALCON compiler, as well as in the more recent MaJIC just-in-time compiler from the same group, is based on data-flow analysis [2, 1]. As noted earlier, this method is inadequate for the speculative specialization we require. Further, FALCON aims to compile whole applications. Our focus in on automatic library generation where the calling context is not available at compilation time to perform interprocedural analysis.

Jump functions provide summary information that allow the compiler to more easily reason about the flow of values around procedure calls[10]. We extended this idea to include information about types through procedures.

Waveren, et al., developed a library generator for the HPF Library. Their primary goal was to handle the large number of context-specific functions using a template-based approach [16].

Inferring types is a well studied problem in the programming languages community. However, almost all type inference in that community is done in the context of functional languages. Further, type inference carried out in the world of programming languages is usually targeted at proving programs correct and presenting inferred types to the users for possible debugging. We *assume* programs to be correct and require library writers to *annotate* procedures including, possibly, type information about the arguments or return values. Because our goal for type inference is to produce type jump functions and generate variants, we are not only interested in the possible types for each variable, but the surrounding context necessary for those types to occur.

A strategy that has proved useful for arrays is the use of dependent types for checking array bounds [18]. The approach of Xi and Pfenning is also based on constructing boolean constraints and solving them. However, we have a somewhat different task of inferring array sizes in the presence of very general types of annotations and also, potentially, performing part of the inference at runtime.

In the functional world, type-based specializations have been used to optimize functional programs [11, 6, 14]. However, none of these seem to have used the kind of comprehensive annotation-based approach to speculatively specialize libraries within the context of an elaborate telescoping-languages system as is proposed in this work.

ATLAS and FFTW are examples of specialized libraries that are automatically tuned for specific platforms [17, 9]. In this sense, the ideas behind ATLAS or FFTW are similar, albeit complementary, to those proposed by telescoping languages. Telescoping languages works well in conjunction with ATLAS, and in fact, we use ATLAS-tuned BLAS routines to handle all the matrix/vector computations. In this way, our compiler is specialized for the different possible platforms.

5. CONCLUSION

We motivated the idea of speculative specialization of libraries based on types. This 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 can induce a specialized variant given enough optimization opportunity. In practice, the number of variants can be limited by annotations from the library writer.

Our study of the ARPACK linear algebra library demonstrates the value of speculative type-based specialization. Speculation allows us to generate highly optimized specialized variants at the library compilation (language generation) time. We demonstrated our technology on a subroutine from ARPACK and showed that we could generate a mathematically equivalent version from Matlab code. Ultimately, this will allow the ARPACK developers (as well as library developers in general) to maintain their code entirely in Matlab while still taking advantage of the performance of Fortran or C, which will be a big step towards our long-term goal of increasing programmer productivity.

6. 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] A. Chauhan. Telescoping MATLAB for DSP applications: Thesis proposal. Technical Report TR02-409, Rice University, 2002.

[4] A. Chauhan and K. Kennedy. Procedure strength reduction and procedure vectorization: Optimization strategies for telescoping languages. In *Proceedings of ACM-SIGARCH International Conference on Supercomputing*, June 2001.

[5] 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.

[6] O. Danvy. Type-directed partial evaluation. In *Proceedings of ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages, St. Petersburg, Florida*, pages 242–257, Jan. 1996.

[7] 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.

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

[9] 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.

[10] 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.

[11] C. V. Hall. Using Hindley-Milner type inference to optimise list representation. In *Proceedings of the 1994 ACM conference on LISP and functional programming*, pages 162–172. ACM Press, 1994.

[12] 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.

[13] C. McCosh. Type-based specialization in a telescoping compiler for MATLAB. Master's thesis. Technical Report TR03-412, Rice University, 2003.

[14] A. Ohori. Type-directed specialization of polymorphism. *Information and Computation*, 155(1-2):64–107, 1999.

[15] D. C. Sorensen, R. B. Lehoucq, and C. Yang. *ARPACK Users' Guide: Solution of Large Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*, 1997.

[16] M. van Waveren, C. Addison, and P. Harrison. Code generator for the HPF library and Fortran 95 tranformational functions. *Concurency–Practice and Experience*, 14(8–9):589–602, 2002.

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

[18] 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.