Type-Based Speculative Specialization in a Telescoping Compiler for Matlab

Arun Chauhan Cheryl McCosh Ken Kennedy
Department of Computer Science, Rice University, 6100 Main Street, Houston, TX

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. We show that type-based specializations of generated libraries can lead to more than 50% speedup.

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 develops an approach combining static and dynamic type inference that includes a graph-theoretic algorithm that is shown to be efficient under a set of conditions that are easily met for most practical cases. The ideas developed here provide a basis for building a more general telescoping system for Matlab [22].

1 Introduction

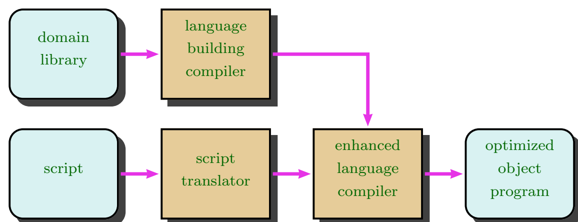

Figure 1: Overview of the telescoping languages approach.

The telescoping languages strategy is aimed at providing end-users interested in scientific applications the ability to write high performance applications in high-level languages [20]. The key idea in achieving this is an extensive offline processing of libraries to speculatively create *specialized variants* that have been optimized for specific calling contexts. The generation of variants is driven by *annotations* by the library writer as well as compiler analysis. The approach has been designed to optimize numerical scientific applications, written in high-level languages, and is based on the observation that such programs spend almost all their computation time inside libraries. Figure 1 depicts the idea graphically.

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. Once the prototype was completed, he and his student painstakingly translated it into several different Fortran variants, specialized to matrix class and data type, to achieve high performance. For example, different Fortran versions were generated based on whether an input array was complex or real, or whether the array was symmetric or not. Software engineering limits, imposed by the amount of programming effort involved, prevented other desirable type-based specializations from being implemented.

This observation about ARPACK motivated us to undertake the development of *ARGen*—a telescoping library generator for ARPACK driven by type-based procedure specialization. Developers of ARPACK had a clear idea of the types anticipated in the use of their libraries, which guided them in translating the Matlab procedures into Fortran. The same guidance provided to a library generator in the form of *annotations* could make it possible to generate specialized variants of the library procedures automatically and *speculatively*, for a variety of anticipated calling contexts.

In order to generate type-based specializations of a procedure, we must be able to infer Matlab variable types. In particular, ARGen must be able to construct a *type jump function* that maps the types of input parameters to a procedure to the types of output variables. These type jump functions would be used in optimizing other libraries, or end-user scripts, that call the library procedure. This paper evaluates the utility of specializing libraries based on variable types and describes a novel technique, combining static and dynamic analyses, to infer the type information of Matlab variables. The technique includes a static algorithm

that is efficient under assumptions that are reasonably met for practical programs. The algorithm is able to discover all valid configurations of variable types, paving the way to generate specialized variants in the telescoping languages framework. The number of variants can be pruned under the directions of annotations by the library writer.

We start with a brief description of the ARPACK linear algebra library.

2 ARPACK

ARPACK stands for ARnoldi PACKage [25]. It is a collection of Fortran 77 subroutines designed to solve large-scale eigenvalue problems. ARPACK implements a variant of the Arnoldi Process called IRAM. The Fortran subroutines are specialized based on types `XY` where `X` can be:

- single precision real arithmetic,
- double precision real arithmetic,
- single precision complex arithmetic, or
- double precision complex arithmetic;

and Y can be:

- non-symmetric, or
- symmetric.

These types occur frequently and have the greatest need for specialization. The developers of ARPACK saw the benefit and necessity of having specialized subroutines based on type, which motivates the work in this paper.

3 Matlab Types

In order to carry out specializations based on variable types we need to first define the variable properties that are of interest to us. In general, the properties should be such that they can be encoded in the target language and the compiler for that language is able to leverage that information for optimization. Since the focus of this work is on numerical applications that are highly oriented towards array manipulations we use the 4-tuple definition of a variable *type* used by deRose [11]. We define a type to be a tuple $\mathcal{T} = <\tau, \rho, \sigma, \psi>$ where,

τ is the intrinsic type of the variable (e.g., integer, real, complex);

ρ is the upper bound on the number of dimensions for an array variable, also called the rank;

σ is the size of an array variable (undefined for a scalar), in turn a tuple of size ρ; and

ψ is the "shape" of an array variable (undefined for a scalar) (e.g., dense, triangular, symmetric, etc.).

3.1 Preliminary Background

We restrict ourselves to a core of the Matlab language that is used by an overwhelming majority of scientific applications. This excludes dynamic evaluation of strings as code, the rather primitive object oriented features, passing function handles as arguments, and structured types (although, the last can be handled by a very simple extension of our ideas).

To avoid coincidental sharing of names among unrelated values, and hence types, of variables we assume that the procedure has been SSA transformed [8]. 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.

The type of a variable depends on:

1. the operation that *defines* the variable, since in the SSA form there is exactly one definition for each variable; and
2. all the operations where a variable is used, since each operation imposes certain restrictions on the types of values it can accept.

The first causes *forward propagation* of variable properties along the control flow, while the second causes *backward propagation*. Formulating this problem as a dataflow framework is not adequate for our purposes. Since a procedure call can cause the type of a variable to be modified in arbitrary ways, and since the control flow graph needs to be traversed in both directions, a conventional dataflow solver is not guaranteed to terminate. Further, the lattice induced by the size of an array (σ) is infinite, with infinite chains over a meet operation defined to be the max operation over non-negative integers. This forces ad-hoc workarounds into any dataflow solver built over it. Finally, we are interested in obtaining all possible configurations of valid types for variables to trigger specialization. A dataflow solution would be either too conservative or not produce all possible values.

We propose using *type jump functions* akin to those used in interprocedural compiler analyses [4, 15]. These type jump functions summarize the transfer functions for library procedures but restrict themselves to the types of the input and output values. They are also used to summarize properties of the primitive operations. In high-level languages operations are invariably syntactic sugar for library calls to the runtime system and a major goal of the telescoping languages strategy is to eliminate the difference between

```
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 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. The scope of a $ variable is the surrounding conjunction (clause).

the two. All operations or procedures used in the procedure being compiled are assumed to have been analyzed and summarized.[1]

3.2 Propositional Formulation

An alternative to formulating the problem in a dataflow framework is analyzing the whole procedure simultaneously using propositional logic. The compiler determines the information that each individual operation or procedure call gives about the variables involved and then combines that information over the entire procedure.

The information from the operations is given in the form of *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 provide all possible type configurations on the variables involved. The 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, "`*`".

The constraints are formed using a database of *annotations* containing one entry per procedure or operation. These annotations could either be derived out of the type jump functions from a previous phase of the library analysis or directly from library writer's annotations, especially for the cases where the source may not be available.

In a correct program, the type of a variable must satisfy all the constraints on it imposed by all the operations that can be feasibly executed in any run of the program. This is equivalent to taking a conjunction of the constraints and finding all possible type-configurations of variables that satisfy the resulting boolean expression. It turns out that this problem in the context of telescoping languages is a hard one to solve even for straight line code. It is, in fact, $\mathcal{NP}$-hard.[2] However, under certain conditions that occur most frequently in practice, we can devise an efficient algorithm to solve the problem. Section 4 describes such an algorithm.

4 An Efficient Algorithm

We will start by describing the general algorithm for inferring types, and then discuss how it can be applied to inferring types in Matlab in the subsequent section.

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

1. The compiler has correct code on input.[3] Although in some cases the compiler may be able to determine that a procedure is incorrect (i.e., if it proves that the whole-procedure constraints cannot be satisfied), proving correctness is not the responsibility of the compiler.

2. The number of input and output parameters in each operation or procedure is bounded by a small constant. This is important for the complexity of the algorithm to remain small. This is a reasonable assumption, since parameter lists do not grow with the size of the procedure [7].

3. All global variables have been converted to input and output parameters. Because of the previous assumption the number of global variables must be small. Well-written libraries rarely use global variables, so this is not a major obstacle.

4. To form the operation constraints, the algorithm requires the compiler to already have annotations, described in the previous section, on all the operations and procedures called in the procedure. The annotations must be formulated so the clauses in each annotation are mutually exclusive.

4.1 Reducing to the Clique Problem

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

Figure 3 shows 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 that

[1] Recursion is handled later.

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

[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.

`A = b + c`

1a $\sigma^A = <1, 1>$ & $\sigma^b = <1, 1>$ & $\sigma^c = <1, 1>$ |
1b $\sigma^A = <\$1, \$2>$ & $\sigma^b = <1, 1>$ & $\sigma^c = <\$1, \$2>$ |
1c $\sigma^A = <\$1, \$2>$ & $\sigma^b = <\$1, \$2>$ & $\sigma^c = <1, 1>$ |
1d $\sigma^A = <\$1, \$2>$ & $\sigma^b = <\$1, \$2>$ & $\sigma^c = <\$1, \$2>$ |

`E = c - d`

2a $\sigma^E = <1, 1>$ & $\sigma^c = <1, 1>$ & $\sigma^d = <1, 1>$ |
2b $\sigma^E = <\$3, \$4>$ & $\sigma^c = <1, 1>$ & $\sigma^d = <\$3, \$4>$ |
2c $\sigma^E = <\$3, \$4>$ & $\sigma^c = <\$3, \$4>$ & $\sigma^d = <3, 1>$ |
2d $\sigma^E = <\$3, \$4>$ & $\sigma^c = <\$3, \$4>$ & $\sigma^d = <\$3, \$4>$ |

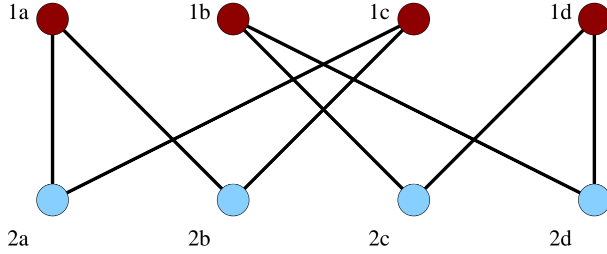

Figure 3: Example graph.

corresponds to its operation number. There is an edge from one node to another if the expressions in the nodes do not contradict one another. The only variable that appears in both operations in the figure is c; therefore, c determines if there is an edge from a node in one level to the next. Without the presence of c, the graph would be complete.

The final graph has n levels, where n is the number of operations or procedure calls. Each level has at most k nodes, where k is l^v and 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. l^v is the number of possible type configurations for each variable in the operation corresponding to that level, since there are l possibilities for each variable. Since by definition, the clauses are mutually exclusive, there are no edges between nodes on the same level.

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 because otherwise the set of clauses would 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) such that each cliques has one and only one node from each level. These conditions are trivially met because there are no edges between nodes on the same level.

4.2 Description of the Algorithm

To describe the basic algorithm, we start with the simplest case and assume the compiler is only analyzing straight-line code where the type of each variable in each operation can be determined by the types of the other variables in the operation (i.e., no data dependence in inferring types). In subsequent sections, we will expand the algorithm to handle the general case.

Since the compiler is going to separately solve the equations in each clique, we need to show that the number of cliques is manageable. Also, since the compiler can potentially generate a variant for each clique, a large number of possibilities would cause a blow-up in the number of variants. We need to 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.

Claim 4.1 *In the absence of control flow with all variables defined in terms of other variables, the number of possible configurations of types is bounded by l^p, where p is the number of input parameters and l is the size of the type lattice.*

Proof: Since the types of all the variables are determined by the types of other variables, all the types should ultimately depend on the types of the inputs. Therefore, the total number of possible configurations over all the variables is just the number of possible configurations of the input parameters. This is l^p since each parameter could take on l types. □

Claim 4.2 *Given the previous claim, the number of n-cliques is bounded by l^p.*

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 at least one level. Since the expressions in nodes of the same level contradict each other, at least one variable in the operation corresponding to that level must have a distinct type. Therefore, the two cliques cannot have the same type assignment to the variables. □

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.

The solution must take advantage of the specific properties of the problem. Figure 4 gives an iterative algorithm for solving the problem. Figure 5 demonstrates the algorithm on an example graph. In each step, the lighter nodes and edges are part of one or more cliques.

The complexity of this algorithm is still exponential in the worst case, since the loop starting on line 4 in figure 4 could iterate over an exponential number of cliques from the previous step. With a limit on the number of cliques

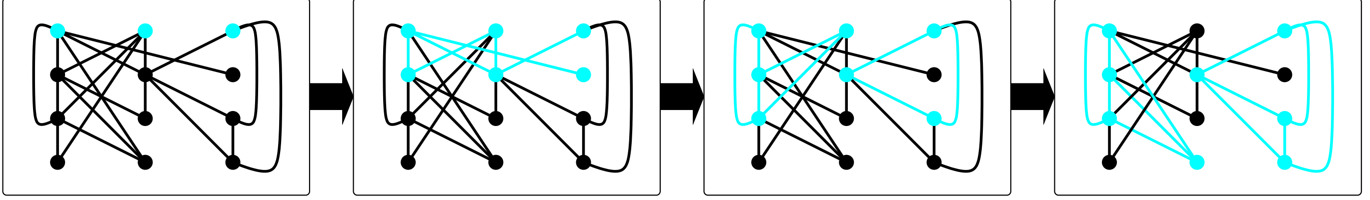

Figure 5: Example of clique-finding algorithm. Each row represents a level in the graph.

```
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.

at each step, the complexity is also limited. Because the bound of l^p only holds for the final number of cliques, we need to find a bound on the number of intermediate cliques to attain this limit. We, therefore, use the structure of the problem to prove a bound of l^p at all intermediate steps if the levels are visited in program order.

Claim 4.3 *Claim: The number of cliques at each step of the clique-finding algorithm is bounded by l^p if the levels are in program order.*

Proof: We have from above that on valid procedures (i.e., procedures whose variables are defined before they are used) there is an upper bound of l^p 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 the algorithm is iterative and only finds cliques over the levels it has already seen, if the algorithm goes in the order of the operations, after every iteration, the algorithm will have produced cliques on valid code. The number of cliques after every iteration must be bounded by l^p. □

With l^p cliques after every iteration, the algorithm takes kl^pn^2 steps. where n is the number of operations, k is the maximum number of nodes in a level, and p is the number of input and output variables per operation. Therefore, the overall time complexity is $O(n^2)$.

Each of the cliques represents a different type configuration. However, the types of the variables satisfying each clique have still not been determined. The equations in each clique need to be solved to find the type allocations. The compiler solves the equations using standard methods in terms of the input parameters so that at runtime, the types of all the variables can be easily inferred from the additional information. However, it may be possible for the compiler to infer exact types for some or all of the inputs as well.

4.3 Annotation Issues

The algorithm assumes that any called procedure has already been analyzed. If it encounters one for which the source code is not available and, therefore, has no annotations, it simply ignores the call as it does not give any added information. This degrades the analysis of the algorithm, although it does not affect the correctness. In essence, it says that the variables are unconstrained by the operation. Recursive calls are treated as unanalyzed procedure calls. While following the type information from the recursion to a fixed point could lead to more exact information, we leave this idea to be explored in future research.

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.

We assume the user-defined annotations concerning types are in the same form as the operation constraints. The compiler treats the user-defined annotations as constraints on the procedure header, which corresponds to the zeroth level in the graph. Any cliques occurring in the graph are forced to have part of the user-defined annotations as one of their nodes. This can greatly reduce the number of possi-

ble cliques and, therefore, specialized variants the compiler would have to generate, and in the case of shape, could allow for finer optimization.

4.4 Control Flow

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 straight-line algorithm described in section 4 is easily extended to handle new names introduced by the ϕ functions. In terms of the algorithm, the ϕ node operations do not constrain the variables types involved. Therefore, if the variable defined by a ϕ function is never used it never appears in any constraint and does not affect the running time of the algorithm. This means that we can get the benefit of working with pruned SSA without requiring the code to be in the pruned form.

The outcome of the ϕ nodes can affect sizes of variables in the procedure when the ϕ variable is used. Therefore, since the outcome of the ϕ operation may not be known until runtime, variables defined by ϕ functions are treated as input arguments to the procedure being analyzed. The uses of the variable defined by the ϕ function may allow the algorithm to determine the outcome of the ϕ function.

In the presence of control flow, the number of possible cliques increases to l^{p+c} where c is the number of ϕ functions in pruned SSA form of the code. However, in most scientific applications, the number of ϕ nodes is usually small, and the procedures written in Matlab are also generally small, so this should not be a serious limitation to the algorithm.[4] The algorithm still behaves correctly regardless of the number of ϕ nodes.

The addition of control flow means a single input configuration can map to multiple cliques. Therefore, the compiler is not able to generate a specialized variant of the entire procedure for the added cliques corresponding to decisions from ϕ nodes. However, the compiler can generate specialized paths from the control flow points 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 keeping the paths separate could allow for optimization opportunities. Since this could cause an increase in the size of the code, the compiler would have to be careful about how many specialized paths are generated. An alternative would be to allocate the meet of the possible types to the variable defined by the ϕ node at the join point. There is a trade-off introduced with this between reducing the number of copies and having the tightest information for optimization. Ultimately, the compiler could just allocate the meet of the types to the variable on all paths.

[4] The procedures from the Matlab development code for ARPACK were typically under fifty lines of code.

4.5 The Result

The compiler ends up with a set of possible variable type configurations over the analyzed procedure. For each possible configuration of types for the input parameters, the compiler creates an individual specialized and optimized variant. The appropriate variant will be linked directly with the user script at script compilation time. In essence, the compiler is producing type jump functions.

In generating the optimized variants, the compiler is able to merge arrays treated as distinct by SSA if the compiler infers that this is beneficial. The compiler can allocate a single variable to have the meet of the types of all the variables being merged. This avoids wasteful copying. However, if the compiler can achieve better performance through optimization by treating variables separately, it does not merge them.

Now that the compiler has information about the inputs and outputs to the analyzed procedure, it computes type jump functions for the database.

5 Inferring $\mathcal{T}$

The algorithm described in the previous section can easily be applied to inferring Matlab types. 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 ρ and σ, the types are independent of each other.

5.1 Inferring Rank and Size

Inferring sizes is slightly different from inferring the rest of the types, since the problem is described by two lattices. Because operators in Matlab have different meanings depending on whether they are operating on scalars or arrays, the primary information needed for inferring sizes is whether the variables are scalars or arrays. Therefore, the $ variables appearing the size constraints not only serve as place-holders, but also determine whether the variable is a scalar or an array. If a size that is not 1 appears in one of the fields of σ for a variable in the constraint, it is assumed by the compiler that that variable cannot be a scalar. The constraints and annotations must enumerate all possible scalar versus array configurations. This means $l = 2$ for inferring sizes in the algorithm. The solver determines the actual values of the $ variables in terms of the unknowns, and therefore works on the infinite integer lattice, where the meet operation produces the maximum size. Forming

constraints for the size-inference problem has already been shown in section 3.

5.1.1 Dimensions

Recall from Section 3 that σ is a tuple consisting of ρ fields, where each field is the size of the variable in the corresponding dimension and ρ is an upper bound on the number of dimensions. In order to infer σ, the compiler needs to first determine ρ. The compiler only needs ρ to be an upper bound, since size-inference will be able to tighten the number of dimensions by inferring that certain dimensions have size 1.

To get the ρ information, the compiler performs a single prepass over the code to see which dimensions are accessed in which variables, either by direct subscripted accesses or by operations. Some operations also require that the number of dimensions for the variable be limited.

When the prepass cannot determine a bound, the compiler creates a dummy dimension field in σ for the variable, representing all dimensions that may behave differently from the rest. This handling of the extra dimensions is valid since they are not accessed explicitly. Therefore, they must have identical behavior.

5.1.2 Subscripted Array Accesses

When only part of an array is accessed in an operation, there are no constraints on that array for that dimension. The sizes of the other variables are, however, constrained by the size of the part of the dimension accessed.

If the size of the array access is defined in terms of the value of another variable, the compiler needs to account for the fact that the value could make the access scalar. Figure 6 illustrates how the constraints are written to handle this situation. The highlighted portions do not appear in the actual constraint, but are left in to show the relationship of the other variables to the piece of v accessed.

```
w = A * v(i:j,:)
```

$\sigma^w = <1, 1>$ & $\sigma^A = <1, 1>$ & $\sigma^{v(1:j,:)} = <1, 1>$ |
$\sigma^w = <j, \$1>$ & $\sigma^A = <1, 1>$ & $\sigma^{v(1:j,:)} = <j, \$1>$ |
$\sigma^w = <\$1, \$2>$ & $\sigma^A = <\$1, \$2>$ & $\sigma^{v(1:j,:)} = <1, 1>$ |
$\sigma^w = <\$1, \$2>$ & $\sigma^A = <\$1, j>$ & $\sigma^{v(1:j,:)} = <j, \$2>$ |
$\sigma^w = <1, 1>$ & $\sigma^A = <1, j>$ & $\sigma^{v(1:j,:)} = <j, 1>$

Figure 6: Handling subscripted array access.

Notice, that the resulting data dependencies could increase the number of cliques and therefore, the complexity since the value of the variable may not be known at library compilation time. The procedures in Matlab are typically small so this should not cause a problem in practice. However, if the compiler can determine that the subscript size is greater than one, then it can add a constraint that forces the variable to be non-scalar, reducing the number of cliques. The algorithm still works correctly if the complexity increases.

5.2 Intrinsic Types

The constraints on intrinsic types are similar to the constraints on size except that instead of working with infinite numbers, the compiler operates on the intrinsic type lattice. However, the lattice for intrinsic types is bigger (for Matlab, there are 6 elements in the intrinsic type lattice), and therefore, l is bigger. In practical cases, the number of possible intrinsic types involved in operations should be smaller than the entire lattice.

The constraints track the range of possible intrinsic types for the variable on the lattice of possibilities. For example, an input argument that is defined as type `real` could actually be of type `int` when called.

For the operation, $A = B + C$, some of the constraints are:

...
$(\texttt{int} \leq \tau^A \leq \texttt{int})$ $\&(\perp \leq \tau^B \leq \texttt{int})$ $\&(\perp \leq \tau^C \leq \texttt{int})|$
$(\texttt{real} \leq \tau^A \leq \texttt{real})$ $\&(\perp \leq \tau^B \leq \texttt{real})$ $\&(\perp \leq \tau^C \leq \texttt{real})|$
...

Two constraint clauses are contradictory if a variable is in both clauses and the ranges of possibilities for the variable's intrinsic type do not intersect.

Once the compiler has found the cliques, solving the equation within the cliques corresponds to taking the intersection of all ranges for each variable.

5.3 Shape

Inferring shapes is similar to inferring intrinsic types. For shapes, however, the compiler needs to rely much more heavily on the annotations. Because the compiler cannot generate a specialized version for every possible shape, the compiler simply generates the most general cases unless it can infer that more specific cases are important either from constraints, or from the user-defined annotations.

The shape lattice includes all possible combinations of shape with $\top$ being the most general (dense) and $\perp$ meaning that the inferred shapes were infeasible. Some examples of entries in the lattice include, sparse, sparse banded, sparse symmetric, sparse symmetric banded, etc.

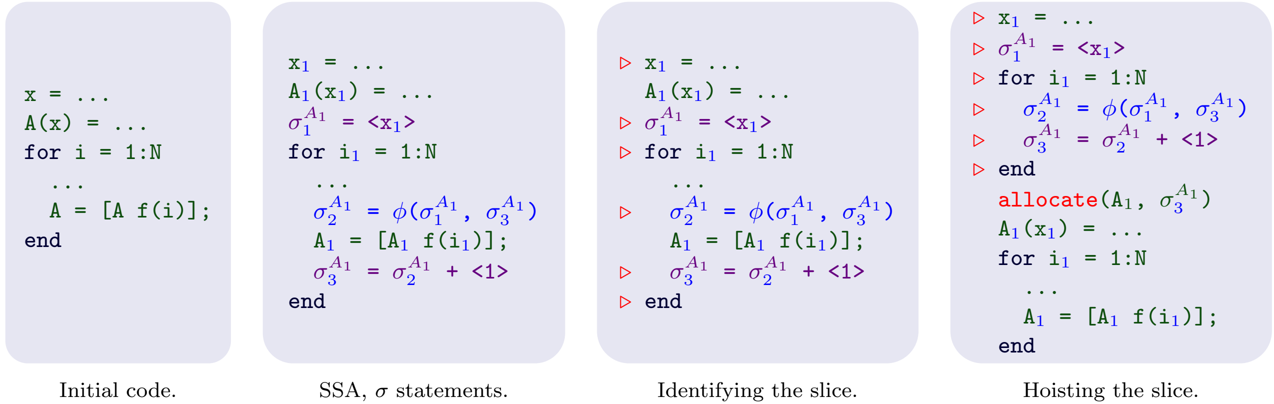

Figure 7: Slice hoisting applied to loops.

6 Slice Hoisting

Array size has the unique characteristic that it may be computed at runtime, when it is not possible to compute it statically. As long the size can be computed before the first reference to the array no copying overhead need be incurred since the array may be allocated dynamically.

In some cases, dynamic inference is unavoidable. For example, the clique-finding algorithm could result in symbolic values for sizes that are not known until runtime. Another case when this happens is in a statement such as `A(i,j) = b`. The clique-finding algorithm ignores the left hand side, however, Matlab semantics would cause the array to be resized if the indices exceed the current size. In both of these examples, σ depends on data values.

Control flow introduces ϕ functions in the SSA form. Static size inference based in the algorithm of section 4 ignores the meet induced by the ϕ functions. In some cases this can lead to incomplete inference forcing the compiler to insert copy statements to preserve correctness, which can be expensive for arrays.

To address these issues, we introduce a source level program transformation called *slice hoisting*. Suppose we denote the size of an array `A` by a pseudo-variable, σ^A. After every statement that could potentially alter `A`'s size, we add a pseudo-statement expressing σ^A in terms of the current σ^A and other known values. This is always possible since the array's size must be computable at runtime. Thus, for the example statement earlier in this section we add the statement σ^A `= max(`σ^A`, <i, j>)`. Next, we perform an SSA renaming of these pseudo-statements to assign a unique name to every definition of the σ variables. This keeps track of the current size at each reference of the array that is needed to generate correct code. Then, we identify the part of the code, called a *slice*, that computes the σ values for any given array. The slice could consist of control structures, including loops. Finally we *hoist* the slice up to before the first reference to the array in the procedure. The hoisting process could involve splitting a loop or duplicating a control structure. Figure 7 shows an example of the application of this technique to a loop.

At the end of this four-step process all the code that was involved in determining the size of the array has been moved to before the first reference to the array. Therefore, the array can be dynamically allocated at this point based on the computed size. Often the hoisted slice can be evaluated completely statically, or replaced by a simple symbolic value, based on constant propagation. Induction variable analysis can also produce symbolic or constant values for variables computed inside loops [1]. In other cases, the resulting code resembles an "inspector-executor" style computation of size.

There are some rare cases when dependences can prevent a slice from being hoisted. For example, when an array's value is used to change its size. In these situations the array must be resized dynamically. Notice that the technique for slice hoisting uses dependence information but does not rely on its accuracy as long as it is conservative. In fact, a simple SSA based dependence testing suffices for most cases. If a more sophisticated dependence analysis is available the technique automatically leverages it by enabling hoisting in more cases.

Adding pseudo statements takes one pass over the program and the ϕ functions. The time to identify slices is proportional to the size of the SSA or the dependence graph. Replicating control flow could lead to, in the worst case, a doubling of the code size. In practice, this rarely happens.

7 Implementation Strategy

We are now ready to describe the implementation strategy for type-based specialization. Figure 8 outlines the steps

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 8: Implementation strategy for ARGen.

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

MathWork Inc.'s `mcc` compiler generates C code that resembles a parse tree of the Matlab code where every primitive operation has been replaced by the call to a runtime library procedure. There are software engineering benefits to starting with this C code, instead of Matlab code. It not only guards against the proprietary incremental developments of Matlab but also provides a uniform way to handle primitive operations as well as user or domain-specific procedures. Another advantage is that we start with C code that can be compiled and run correctly with any standard C compiler. The generated code is in C, or equivalent Fortran, with the calls to relevant specialized variants replacing the calls to generic procedures wherever specialization succeeds. The remaining cases, if any, are automatically handled by the generic procedures. In this way, calls to the runtime library for primitive operations, as well as to any domain-specific or user defined library, can be handled uniformly.

For a proof-of-concept implementation we have built a system for inferring array sizes that parses Matlab directly. The system reads the type jump functions, encoded as annotations in a propositional logic form, and builds constraints for all the operations and procedure calls in the given Matlab procedure. Finally, it builds a graph, discovers cliques using the algorithm in section 4, and solves the cliques to determine array sizes. The solver in the current form uses a simple substitution based algorithm, however, there are known techniques to build more sophisticated solvers.

8 Experimental Evaluation

ARPACK libraries contain certain details that are not handled by the proof-of-concept implementation. Therefore, we carried out the type inference process and generated Fortran code by hand, carefully performing only the steps that the compiler is expected to perform.

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

Figure 9 shows the SSA form of the Matlab ArnoldiC—a procedure from ARPACK—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 on the input parameters. It turns out that 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.

Based on the configuration obtained above, we manually generated Fortran code that used ATLAS-tuned BLAS for matrix operations to simulate the code generated by ARGen. The simulated code was specialized on shape (symmetric and non-symmetric) and intrinsic type (real and complex) and compiled with `-O3` optimization flag. In all cases, the input matrix, A_1 was the same real, sparse, array from Matrix Market with 3074 entries [21]. k_1 was set to 30 and v_1 to be a 362-length vector. These sizes conform to the requirements of configuration C in figure 9. We

used version 6.1 for Matlab, which also uses ATLAS-tuned BLAS at the bottom level. The runtimes were measured on a 143 MHz SPARC processor.

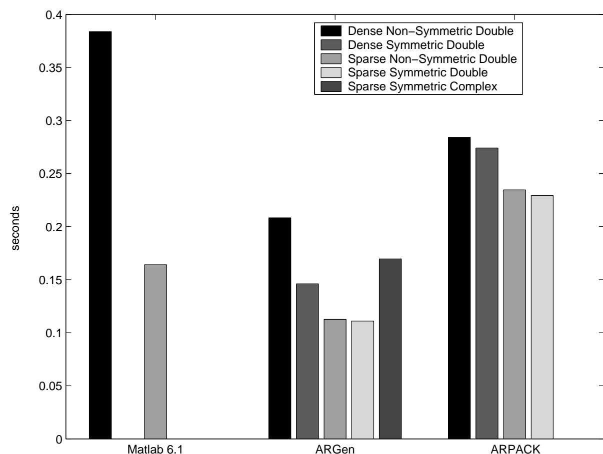

Figure 10: ArnoldiC running times: Matlab, simulated ARGen, and Fortran ARPACK.

Figure 10 shows a comparison of running times for the original Matlab code and the hand-simulated versions. We included the runtimes for Fortran ARPACK (`dnaitr` and `dsaitr` subroutines) in order to demonstrate that the running times are in the same order of magnitude. The times for Fortran ARPACK are not directly comparable since they behave slightly differently from the Matlab version of the code as the ARPACK developers added extra functionality to it when rewriting the code in Fortran.

The graph shows the value of type-based specialization. There is an almost 50% speedup in going from a dense non-symmetric representation to a sparse symmetric representation. This illustrates the importance of inferring the right shape for a matrix, and using specialized library procedures for that shape. Similarly, specializing on the intrinsic types has a high payoff as seen from a 54% speed improvement going from using libraries operating on complex types to those specialized on real floating point numbers.

Finally, the performance gains in going from Fortran to simulated ARGen reinforce the benefits of compiling Matlab into a lower-level language.

Most other Matlab procedures in ARPACK are very similar to ArnoldiC and we expect them to benefit from the same type-based specializations. While slice-hoisting was not applicable to ArnoldiC because all the variable sizes could be inferred statically, many of the routines in ARPACK have variable sizes that have control dependences, where slice-hoisting will be valuable in performing dynamic size inference.

9 Related Work

The FALCON compiler at the University of Illinois carried out type inference for Matlab for generating code in Fortran 90 [11, 10]. The 4-tuple definition of types 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 dataflow analysis [3, 2]. As noted earlier, this method is inadequate for the speculative specialization we require.

Inferring types is a well studied problem in the programming languages community. 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. In the functional world, type inference usually does not treat numerical quantities or arrays in great detail. Matlab, on the other hand, is an imperative language that is heavily oriented towards array manipulation. We are not interested the most general type of a variable, but all possible configurations of types of all the variables that would allow the program to execute correctly in order to generate specialized variants.

In spite of these broad differences it is instructive to compare this work to some typical pieces of type inference work in programming languages community.

The well known Hindley-Milner type systems can be solved using standard unification-based algorithms [23]. These systems include a pure form of polymorphism. However, pure polymorphism is not sufficient to express the type system of Matlab [5]. Since annotations could be arbitrary, the types need, in Cardelli's words, "bounded quantification". Some recent work has explored combining constraints with the traditional type inference methods to handle more elaborate type systems in the context of functional languages (for example, see [27]). However, we do not know of any application of these systems to handle indexed data structures, such as arrays.

Flanagan used componential set-based analysis to infer types for the purposes of debugging [13]. However, his system does not handle the heavily overloaded operators found in Matlab.

A strategy that has proved useful for arrays is the use of dependent types for checking array bounds [30]. 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, making part of the inference at runtime.

In the functional world, type based specializations have been used to optimize functional programs [16, 9, 24]. 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.

Techniques proposed in this paper can produce code to work with specialized libraries that are automatically tuned for specific platforms or problems, for example, ATLAS and FFTW [29, 14]. For example, specialized FFT routines can be called directly if the input matrix size is known.

Constraint Logic Programming (CLP) has become popular recently [26, 17, 18]. CLP extends the purely syntactic logic programming (typified by linear unification) by adding semantic constraints over specific domains. Using constraints over type domains would fall in the category of Constraint Logic Programming. Some of the well known CLP systems include CHIP [12], CLP($\mathcal{R}$) [19], Prolog-III [6], and ECL^{i}PS^e [28]. While a general purpose CLP system could be employed in solving the constraints within our type inference system our algorithm utilizes the specific properties of the problem domain to operate within a provably efficient time complexity.

10 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 induces a specialized variant. In practice, the number of variants can be limited by annotations by the library writer.

We formulated the inference problem using propositional logic utilizing type jump functions encoded in the form of an annotation database on library procedures. The database is populated by the library compiler either through direct analysis or a transcription of user provided annotations for the procedures that are not analyzed (say, when the source is not available). This formulation enables solving for all possible type combinations simultaneously.

Type inference is carried out in two phases. The first phase maps the constraints onto a graph and uses a clique-finding approach to identify valid type combinations. This phase also determines array ranks and shapes and intrinsic types for all the variables. The second phase handles inference for array sizes, that could be not be handled by the first, using a technique called slice hoisting.

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.

11 Acknowledgements

We thank Prof Dan Sorensen for providing the motivation behind this work and the code for ARPACK. He helped us understand the problem as well as the many issues involved. We thank him for his time and support.

References

[1] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach.* Morgan Kaufmann Publishers, San Francisco, CA, 2001.

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

[3] 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.

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

[5] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, Dec. 1985.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] C. Flanagan. *Effective Static Debugging via Componential Set-Based Analysis*. PhD thesis, Rice University, Houston, Texas, May 1997.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] http://math.nist.gov/MatrixMarket/. Matrix Market.

[22] C. McCosh. Type-based specialization in a telescoping compiler for MATLAB. Master's thesis, Rice University, Houston, Texas, 2002.

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

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

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

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

[27] M. Sulzmann. *A General Framework for Hindley/Milner Type Systems with Constraints*. PhD thesis, Yale University, Department of Computer Science, May 2000.

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

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

[30] 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.

A $\mathcal{NP}$-completeness of Type Inference for Straight Line Code

Operations on variables can impose certain constraints on their types, including their ranks and intrinsic types. In the absence of any other information it may be desirable to generate these constraints at compile time, based upon the properties of the operation, and deduce feasible combinations for the variables. In the telescoping languages context this can serve as a guide to specialize the program or procedure. In a general case this problem is undecidable (halting problem can be easily reduced to it). Even in the more restricted case of straight line code (that has no branches) this is a hard problem. The following theorem states this for array ranks.

Theorem A.1 *Inferring feasible (valid) rank combinations for variables in a program without branches is $\mathcal{NP}$-complete.*[5]

In order to prove the theorem we must show that the problem is in $\mathcal{NP}$ and that it is $\mathcal{NP}$-hard. The problem can be solved in polynomial time by a non-deterministic Turing Machine by simply guessing feasible ranks for the variables and verifying that the constraints imposed by all the statements in the program are satisfied. The verification is easily done in polynomial time for linear code. Thus, the problem is, clearly, in $\mathcal{NP}$.

We reduce 3-CNF SAT to this problem to complete the proof. 3-CNF SAT is the problem of determining whether a satisfying truth assignment to variables exists for a Boolean expression in conjunctive normal form where each clause consists of exactly three literals. The problem statement in 3-CNF is of the form

$$\bigwedge x_{i_1} \vee x_{i_2} \vee x_{i_3}$$

where x_i denotes a literal that could be a variable v or its negation $\bar{v}$.

Given a program statement `f(A,B,C)` there are some constraints imposed on the ranks of `A`, `B`, and `C` depending on the function `f`. For example, the function `f` might impose the constraint that if `A` and `B` both have rank 2 then `C` must also have the rank 2. If we denote the rank of a variable `V` by ρ^V then this translates to the constraint:

$$((\rho^A = 2) \wedge (\rho^B = 2)) \Rightarrow (\rho^C = 2)$$

Using the identity $\alpha \Rightarrow \beta \equiv \neg\alpha \vee \beta$ and de Morgan's law, this reduces to

$$\neg(\rho^A = 2) \vee \neg(\rho^B = 2) \vee (\rho^C = 2)$$

[5] We use $\mathcal{NP}$-completeness in the sense of algebraic complexity, not bit complexity.

operation	rank constraint	clause
$\mathtt{f}_1\mathtt{(a,b,c)}$	$((\rho^a = 0) \wedge (\rho^b = 0)) \Rightarrow (\rho^c = 2)$	$a \vee b \vee c$
$\mathtt{f}_2\mathtt{(a,b,c)}$	$((\rho^a = 2) \wedge (\rho^b = 0)) \Rightarrow (\rho^c = 2)$	$\bar{a} \vee b \vee c$
$\mathtt{f}_3\mathtt{(a,b,c)}$	$((\rho^a = 0) \wedge (\rho^b = 2)) \Rightarrow (\rho^c = 2)$	$a \vee \bar{b} \vee c$
$\mathtt{f}_4\mathtt{(a,b,c)}$	$((\rho^a = 2) \wedge (\rho^b = 2)) \Rightarrow (\rho^c = 2)$	$\bar{a} \vee \bar{b} \vee c$
$\mathtt{f}_5\mathtt{(a,b,c)}$	$((\rho^a = 0) \wedge (\rho^b = 0)) \Rightarrow (\rho^c = 0)$	$a \vee b \vee \bar{c}$
$\mathtt{f}_6\mathtt{(a,b,c)}$	$((\rho^a = 2) \wedge (\rho^b = 0)) \Rightarrow (\rho^c = 0)$	$\bar{a} \vee b \vee \bar{c}$
$\mathtt{f}_7\mathtt{(a,b,c)}$	$((\rho^a = 0) \wedge (\rho^b = 2)) \Rightarrow (\rho^c = 0)$	$a \vee \bar{b} \vee \bar{c}$
$\mathtt{f}_8\mathtt{(a,b,c)}$	$((\rho^a = 2) \wedge (\rho^b = 2)) \Rightarrow (\rho^c = 0)$	$\bar{a} \vee \bar{b} \vee \bar{c}$

Figure 11: Table of operations along with their rank constraints and the corresponding 3-CNF clauses.

Also, notice that to respect the constraints imposed by a linear sequence of such program statements *all* of these constraints must be satisfied. In other words, the constraints must be composed by conjunction.

Suppose that each variable in the program can only have a rank of 0 or 2. In that case, the expression $\neg(\rho^v = 2)$ is equivalent to $(\rho^v = 0)$. For every 3-CNF SAT variable v we define a program variable `v`. The variable v is assigned *true* iff the rank of `v` is 2. Thus, v corresponds to $(\rho^v = 2)$ and $\bar{v}$ corresponds to $\neg(\rho^v = 2)$ or $(\rho^v = 0)$. In order to reduce an instance of 3-CNF SAT into an instance of the problem of inferring feasible rank combinations we write a program that consists of a linear sequence of statements. Each statement corresponds to a clause in 3-CNF SAT and imposes a constraint that corresponds exactly to the clause. Figure 11 defines eight functions and the constraints imposed by each of the functions that correspond to the 3-CNF formula shown in the rightmost column. In a real program these functions could correspond to library routines that impose the indicated constraints on their arguments.

For each 3-CNF clause of the form $x_1 \vee x_2 \vee x_3$ we write a statement of the form $\mathtt{f_i(x_1,x_2,x_3)}$ where the function $\mathtt{f_i}$ is chosen according the table in figure 11. Clearly, this construction is polynomial time and log space. We state, without proof, the following lemma.

Lemma A.1 *A given instance of 3-CNF SAT has a satisfying truth assignment iff the variables in the corresponding program have a feasible combination of ranks.*

The proof of the lemma is obvious. This completes the reduction and, hence, the proof of the theorem. □

Theorem A.1 has another implication. Proving a program correct is undecidable in general. However, if the program has no branches then the problem might seem "easy". The theorem indicates that if we must infer array ranks in the absence of any other information then even this can be a hard problem since for the program to be correct there must be at least one feasible combination of variable ranks.

Corollary A.2 *Proving correctness of linear code (without branches) in the presence of arbitrary operators is $\mathcal{NP}$-hard.*

A similar result holds for intrinsic types and array shapes.

Corollary A.3 *Inferring feasible (valid) intrinsic types or array shapes for variables in a program without branches is $\mathcal{NP}$-complete.*

It should be noted that these results hold only in a generalized case of arbitrary operators. In particular, if operators are not polymorphic or overloaded then the problems can be solved in polynomial time. The results can be seen to hold for any finite domain over which strict constraints can be written. However, in most practical domains, the domain elements are arranged in a lattice and it is, usually, permissible to substitute an element by another of lower value (or vice versa). Thus, most practical operators and functions impose *inequality* constraints on types instead of strict equality. For example, a function that accepts a complex number as an argument can also accept a real or an integer (since the others can be type cast to complex). The hardness result for straight line code does not necessarily hold in such cases.