Slice-hoisting for Array-size Inference in MATLAB

Arun Chauhan and Ken Kennedy

achauhan@cs.rice.edu ken@cs.rice.edu
Department of Computer Science, Rice University, Houston, TX 77005

Abstract. Inferring variable types precisely is very important to be able to compile MATLAB libraries effectively in the context of the telescoping languages framework being developed at Rice. Past studies have demonstrated the value of type information in optimizing MATLAB [4]. The variable types are inferred through a static approach based on writing propositional constraints on program statements [11]. The static approach has certain limitations with respect to inferring array-sizes. Imprecise inference of array-sizes can have a drastic effect on the performance of the generated code, especially in those cases where arrays are resized dynamically. The impact of appropriate array allocation is also borne out of earlier studies [3]. This paper presents a new approach to inferring array-sizes, called *slice-hoisting*. The approach is based on simple code transformations and is easy to implement in a practical compiler. Experimental evaluation shows that slice-hoisting, along with the constraints-based static algorithm, can result in a very high level of precision in inferring MATLAB array sizes.

1 Introduction

There is a growing trend among the scientific and engineering community of computer users to use high-level domain-specific languages, such as MATLAB®, R, Python, Perl, etc. Unfortunately, these languages continue to be used primarily for prototyping. The final code is still written in lower-level languages, such as, C or Fortran. This has profound implications for programmers' productivity.

The reason behind the huge popularity of domain-specific languages is the ease of programming afforded by these languages. We refer to these languages as *high-level scripting languages*. The users programming in scripting languages are usually analytically oriented and have no trouble in writing high-level algorithms to solve their computational problems. These languages provide direct mappings of high-level operations onto primitive operations or domain-specific libraries. Unfortunately, the compilation and the runtime systems of high-level scripting languages often fall far short of users' requirements. As a result, the users are forced to rewrite their applications in lower-level languages.

We and our colleagues at Rice have been developing a strategy, called *telescoping languages*, to address the issue of compiling high-level scripting languages

® MATLAB is a registered trademark of MathWorks Inc.

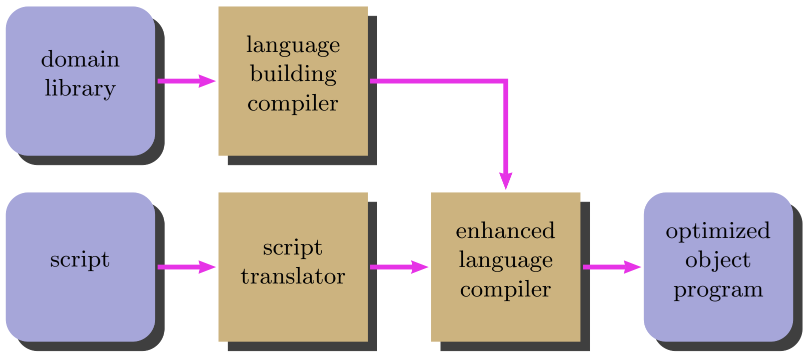


Fig. 1. The telescoping languages strategy

efficiently and effectively [9]. The idea is to perform extensive offline processing of libraries that constitute the primary computation sites in high-level scripting languages. The end-user programs, called the *scripts*, are passed through an efficient script compiler that has access to the knowledge-base built by the library compiler. The script compiler utilizes this knowledge-base to rapidly compile end-user scripts into effective object code. The strategy is outlined in Fig. 1. This strategy enables extending the language in a hierarchical manner by repeating the process of library building, which is the origin of the term "telescoping".

A part of the current effort in telescoping languages is towards compiling MATLAB. The telescoping languages strategy envisions generating the output code in an intermediate language, such as C or Fortran. Emitting code in an intermediate low-level language, rather than the binary, has the advantage of leveraging the excellent C or Fortran compilers that are often available from vendors for specific platforms.

One performance related problem that arises in compiling high-level scripting languages is that of inferring variable types. MATLAB is a weakly typed language, treating every variable as an array and performing runtime resolution of the actual type of the variable. This imposes an enormous performance overhead. It is possible to infer variable types statically, thereby eliminating this overhead [11]. Knowing precise variable types can improve code generation by emitting code that uses the primitive operations in the target language, whenever possible—primitive operations in lower-level languages can be orders of magnitude faster than calling library functions that operate on a generic user-defined type. Earlier studies have found type-based optimizations to be highly rewarding [4].

In order to demonstrate the effect that type inference can have on code generation, consider the Fig. 2 that shows the performance improvements in a Digital Signal Processing procedure, called `jakes`, after it has been translated into Fortran based on the inferred types from the original MATLAB code. No other optimization was performed on this code. There are no results for MATLAB 5.3

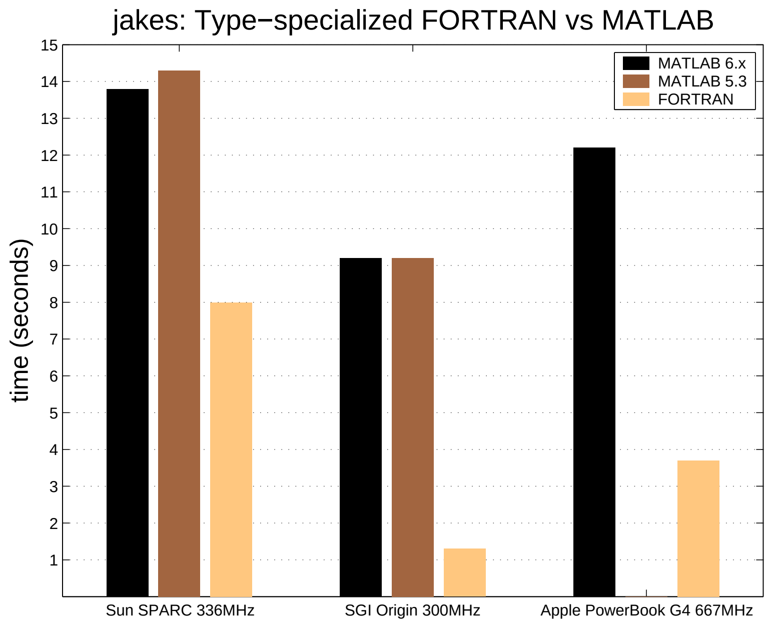


Fig. 2. Value of type inference

on Apple PowerBook because the version 5.3 is not available on the PowerBook. The MATLAB times have been obtained under the MATLAB interpreter. The running times for the stand-alone version that is obtained by converting the MATLAB code into equivalent C using the MathWorks `mcc` compiler are even higher than the interpreted version (possibly because of startup overheads). The "compilation" performed by `mcc` is a very straightforward one in which each variable has the most general possible type and all operations translate to calls to library procedures that can handle any arbitrary argument types. The speed improvements directly show the value of knowing precise variable types.

It is not always possible to do a complete static resolution of variable types, especially, the size of an array that is a component of a variable's type. In such a case the compiler may need to generate code to perform expensive operations such as resizing an array dynamically. We present a new strategy to compute the array-sizes based on slicing the code and hoisting it to before the first use of the array. This strategy can enable resolution of several cases that the static analysis fails to handle. Moreover, slice-hoisting becomes more precise in the presence of advanced dependence analysis while still being useful without it. Experimental results show that slice-hoisting results in substantial gains in the precision of type inference of MATLAB variables for code taken from the domain of Digital Signal Processing (DSP).

2 Types and Array-sizes

MATLAB is an array-processing language. Most numerical programs written in MATLAB rely heavily on array manipulations. Therefore, it is useful to define the type of a MATLAB variable such that the type can describe the relevant properties of an array. A variable's type is defined to be a four-tuple, $<\tau, \delta, \sigma, \psi>$, such that:

τ denotes the intrinsic type of a variable (`integer`, `real`, `complex`, etc.)
δ is the dimensionality, which is 0 for scalar variables
σ is a δ-sized tuple that denotes the size of an array along each dimension
ψ denotes the structure of an array (e.g., whether an array is dense, triangular, diagonal, etc.)

This definition is motivated by de Rose's work [6].

McCosh uses a framework based on propositional logic to write "constraints" for the operands of each operation in a MATLAB procedure being compiled [11]. The constraints are then "solved" to compute valid combinations of types, called *type-configurations* that preserve the meaning of the original program. This process is carried out for each component of the type-tuple defined above. For size (σ), solving the constraints results in a set of linear equations that are solved to obtain array sizes, i.e., the σ values for each type-configuration.

The constraints-based static analysis technique does a good job of computing all possible configurations and taking care of forward and backward propagation of types. However, it has certain limitations.

1. Constraints-based static analysis does not handle array-sizes that are defined by indexed array expressions, e.g., by changing the size of an array by indexing past its current extent along any dimension.
2. The control join-points are ignored for determining array-sizes, which can lead to a failure in determining some array-sizes.
3. Some constraints may contain symbolic expressions involving program variables whose values are not known at compile time. These values cannot be resolved statically.

As a result of these limitations, if a purely constraints-based approach is used to infer array-sizes some of them may not be inferred at all. This can result in generated code that might have to perform expensive array resizing operations at runtime.

Consider the Fig. 3 that shows a code snippet from a DSP procedure. The array `vcos` is initialized to be an empty matrix in the outer loop and then extended by one element in each iteration of the inner loop. At the end of the inner loop `vcos` is a vector of size `sin_num`. The variable `mcos` is initialized to be an empty matrix once outside the outer loop. In every iteration of the outer loop it is extended by the vector `vcos`. At the end of the outer loop `mcos` is a `sin_num` by `sin_num` two-dimensional array. This is a frequently occurring pattern of code in DSP applications and the constraints-based static analysis fails to infer the correct sizes for `vcos` and `mcos`. There is no straightforward way to write constraints that would accurately capture the sizes of `vcos` and `mcos`. The problem here is that the static analysis ignores ϕ-functions, while those are crucial in this case to determine the sizes.

```
for n=1:sin_num
   vcos = [];
   for i = 1:sin_num
      vcos = [vcos cos(n*w_est(i))];
   end
   mcos = [mcos; vcos];
end
```

Fig. 3. Implicit array resizing in a DSP procedure

Past studies have shown that array-resizing is an extremely expensive operation and pre-allocating arrays can lead to substantial performance gains [3, 12]. Therefore, even if the array size has to be computed at runtime, computing it once at the beginning of the scope where the array is live and allocating the entire array once will be profitable.

3 Slice-hoisting

Slice hoisting is a novel technique that enables size inference for the three cases that are not handled by the static analysis:

– array sizes changing due to index expressions,
– array sizes determined by control join-points, and
– array sizes involving symbolic values not known at compile time.

An example DSP code that resized array in a loop was shown in the previous section. Another way to resize an array in MATLAB is to index past its current maximum index. The keyword `end` refers to the last element of an array and indexing past it automatically increases the array size. A hypothetical code sequence shown below resizes the array `A` several times using this method.

```
A = zeros(1,N);
A(end+1) = x;
for i = 1:2*N
     A(i) = sqrt(i);
end
...
A(3, :) = [1:2*N];
...
A(:,:,2) = zeros(3, 2*N);
...
```

This type of code occurs very often in DSP programs. Notice that the array dimensionality can also be increased by this method, but it is still easily inferred. However, the propositional constraint language used for the static analysis does not allow writing array sizes that change. The only way to handle this within that framework is to rename the array every time there is an assignment to any of its elements, and then later perform a merge before emitting code, to minimize array copying. If an array cannot be merged, then a copy must be inserted. This is the traditional approach to handling arrays in doing SSA renaming [5]. Array SSA could be useful in this approach [10].

Finally, if the value of `N` is not known until run time, such as when it is computed from other unknown symbolic values, then the final expression for the size of `A` will have symbolic values. Further processing would be needed before this symbolic value could be used to declare `A`.

Slice-hoisting handles these cases very simply through code transformations. It can be easily used in conjunction with the static analysis to handle only those arrays whose sizes the static analysis fails to infer.

The basic idea behind slice hoisting is to identify the *slice* of computation that participates in computing the size of an array and *hoist* the slice to before the first use of the array. It suffices to know the size of an array before its first use even if the size cannot be completely computed statically. Once the size is known the array can be allocated either statically, if the size can be computed at compile time, or dynamically. The size of an array is affected by one of the following three types of statements:

– A direct definition defines a new array in terms of the right hand side. Since everything about the right hand side must be known at this statement, the size of the array can be computed in terms of the sizes of the right hand side in most cases.
– For an indexed expression on the left hand side, the new size is the maximum of the current size and that implied by the indices.
– For a concatenation operation the new size of the array is the sum of the current size and that of the concatenated portion.

The size of a variable `v` is denoted by σ^v. A σ value is a tuple $<t_1, t_2, \ldots\ t_\delta>$, where δ is the dimensionality of the variable and t_i denotes the size of `v` along the dimension i. The goal of the exercise is to compute the σ value for each variable and hoist the computation involved in doing that to before the first use of the variable. The final value of σ^v is the size of the array `v`. This process involves the following four steps:

1. transform the given code into SSA form,
2. insert σ statements and transform these into SSA as well,
3. identify the slice involved in computing the σ values, and
4. hoist the slice.

These steps are illustrated with three examples in Fig. 4. Steps 1, 2 and 3 have been combined in the figure for clarity. The top row in the figure demonstrates the idea with a simple straight line code. The next row shows how a branch can be handled. Notice that some of the code inside the branch is part of the slice that computes the size of `A`. Therefore, the branch must be split into two while making sure that the conditional `c` is not recomputed, especially if it can have side-effects. Finally, the bottom row of Fig. 4 illustrates the application of slice

(a)

```
A = zeros(1,N);
y = ...
A(y) = ...
x = ...
A(x) = ...
```

```
  $A_1$ = zeros(1, N);
▷ $\sigma_1^{A_1}$ = <N>
▷ $y_1$ = ...
  $A_1$(y) = ...
▷ $\sigma_2^{A_1}$ = max($\sigma_1^{A_1}$,<y>)
▷ $x_1$ = ...
  $A_1$($x_1$) = ...
▷ $\sigma_3^{A_1}$ = max($\sigma_2^{A_1}$,<x>)
```

```
▷ $\sigma_1^{A_1}$ = <N>
▷ $y_1$ = ...
▷ $\sigma_2^{A_1}$ = max($\sigma_1^{A_1}$,<y>)
▷ $x_1$ = ...
▷ $\sigma_3^{A_1}$ = max($\sigma_2^{A_1}$,<x>)
  allocate(A, $\sigma_3^{A_1}$);
  $A_1$ = zeros(1, N);
  $A_1$(y) = ...
  $A_1$($x_1$) = ...
```

(b)

```
y = ...
A(y) = ...
c = ...
if (c)
  ...
  B = [ ... ];
  x = min(B);
else
  ...
  x = 10;
end
A(x) = ...
```

```
▷ $y_1$ = ...
  $A_1$($y_1$) = ...
▷ $\sigma_1^{A_1}$ = <$y_1$>
▷ $c_1$ = ...
▷ if ($c_1$)
    ...
▷   $B_1$ = [ ... ];
▷   $x_1$ = min($B_1$);
▷ else
    ...
▷   $x_2$ = 10;
▷ end
▷ $x_3$ = $\phi$($x_1$, $x_2$)
  $A_1$($x_3$) = ...
▷ $\sigma_2^{A_1}$ = max($\sigma_1^{A_1}$,<$x_3$>)
```

```
▷ $y_1$ = ...
▷ $\sigma_1^{A_1}$ = <$y_1$>
▷ $c_1$ = ...
▷ if ($c_1$)
▷   $B_1$ = [ ... ];
▷   $x_1$ = min($B_1$);
▷ else
▷   $x_2$ = 10;
▷ end
▷ $x_3$ = $\phi$($x_1$, $x_2$)
▷ $\sigma_2^{A_1}$ = max($\sigma_1^{A_1}$,<$x_3$>)
  allocate($A_1$, $\sigma_2^{A_1}$);
  $A_1$($y_1$) = ...
  if ($c_1$)
    ...
  else
    ...
  end
  $A_1$($x_3$) = ...
```

(c)

```
x = ...
A(x) = ...
for i = 1:N
  ...
  A = [A f(i)];
end
```

```
▷ $x_1$ = ...
  $A_1$($x_1$) = ...
▷ $\sigma_1^{A_1}$ = <$x_1$>
▷ for $i_1$ = 1:N
    ...
▷   $\sigma_2^{A_1}$ = $\phi$($\sigma_1^{A_1}$,$\sigma_3^{A_1}$)
    $A_1$ = [$A_1$ f($i_1$)];
▷   $\sigma_3^{A_1}$ = $\sigma_2^{A_1}$+<1>
▷ end
```

```
▷ $x_1$ = ...
▷ $\sigma_1^{A_1}$ = <$x_1$>
▷ for $i_1$ = 1:N
▷   $\sigma_2^{A_1}$ = $\phi$($\sigma_1^{A_1}$,$\sigma_3^{A_1}$)
▷   $\sigma_3^{A_1}$ = $\sigma_2^{A_1}$+<1>
▷ end
  allocate($A_1$, $\sigma_3^{A_1}$)
  $A_1$($x_1$) = ...
  for $i_1$ = 1:N
    ...
    $A_1$ = [$A_1$ f($i_1$)];
  end
```

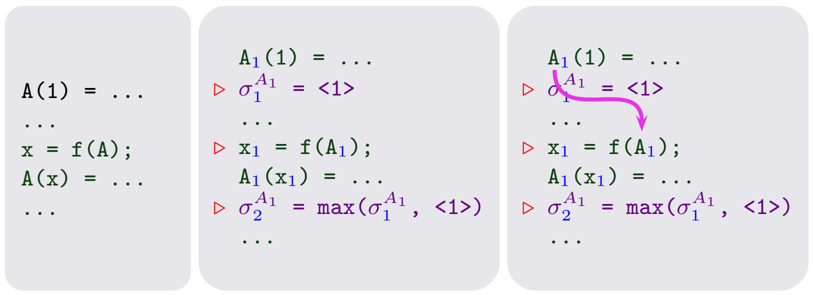
Initial code — Identifying the slice — Hoisting the slice

Fig. 4. Three examples of slice-hoisting

hoisting to a loop. In this case, again, the loop needs to be split. The loop that is hoisted is very simple and induction variable analysis would be able to detect σ^A to be an auxiliary loop induction variable, thereby eliminating the loop. If eliminating the loop is not possible then the split loop reduces to the inspector-executor strategy. Notice that in slice-hoisting concatenation to an array, or assignment to an element of the array, does *not* create a new SSA name.

This approach has several advantages:

– It is very simple and fast, requiring only basic SSA analysis in its simplest form.
– It can leverage more advanced analyses, if available. For example, advanced dependence analysis can enable slice hoisting in the cases where simple SSA based analysis might fail. Similarly, symbolic analysis can complement the approach by simplifying the hoisted slice.
– Other compiler optimization phases—constant propagation, auxiliary induction variable analysis, invariant code motion, common subexpression elimination—all benefit slice hoisting without having to modify them in any way.
– It subsumes the inspector-executor style.
– The approach works very well with, and benefits from, the telescoping languages framework. In particular, transformations such as procedure strength reduction and procedure vectorization can remove certain dependencies making it easier to hoist slices [2].
– Most common cases can be handled without any complicated analysis.



Initial code

```
A(1) = ...
...
x = f(A);
A(x) = ...
...
```

Identifying the slice

$\quad A_1(1)$ = ...
▷ $\sigma_1^{A_1}$ = <1>
$\quad$ ...
▷ x_1 = f(A_1);
$\quad A_1(x_1)$ = ...
▷ $\sigma_2^{A_1}$ = max($\sigma_1^{A_1}$, <1>)
$\quad$ ...

Dependence blocks hoisting

$\quad A_1(1)$ = ...
▷ $\sigma_1^{A_1}$ = <1>
$\quad$ ...
▷ x_1 = f(A_1);
$\quad A_1(x_1)$ = ...
▷ $\sigma_2^{A_1}$ = max($\sigma_1^{A_1}$, <1>)
$\quad$ ...

Fig. 5. Dependencies can cause slice hoisting to fail

In some cases it may not be possible to hoist the slice before the first use of the array. Figure 5 shows an example where a dependence prevents the identified slice from being hoisted before the array's first use. Such cases are likely to occur infrequently. Moreover, a more refined dependence analysis, or procedure specialization (such as procedure strength reduction) can cause such dependencies to disappear. When the slice cannot be hoisted the compiler must emit code to resize the array dynamically.

When slice-hoisting is applied to compute an array size it may be necessary to insert code to keep track of the actual *current size* of the array, which would be used in order to preserve the semantics of any operations on the array in the original program.

4 Experimental Evaluation

To evaluate the effect of slice-hoisting we conducted experiments on a set of Digital Signal Processing (DSP) procedures that are a part of a larger collection of procedures written in MATLAB. The collection of procedures constitutes an informal library. Some of these procedures have been developed at the Electrical and Computer Engineering department at Rice for DSP related research work. Others have been selected from the contributed DSP code that is available for download at the MathWorks web-site.

4.1 Precision of Constraints-based Inference

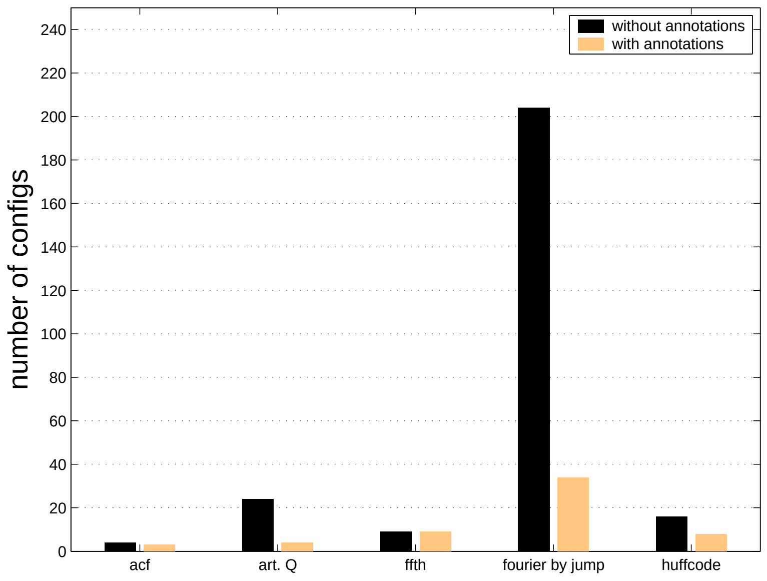


Fig. 6. Precision of the constraints-based type inference

We first evaluated the precision of the static constraints-based type inference algorithm. Due to the heavy overloading of operators, MATLAB code is often valid for more than one combination of variable types. For example, a MATLAB function written to perform FFT might be completely valid even if a scalar value is passed as an argument that is expected to be a one-dimensional vector. The results might not be mathematically correct, but the MATLAB operations performed inside the function may make sense individually. As a result, the static type inference algorithm can come up with multiple valid type-configurations. Additionally, the limitations enumerated earlier in section 2 can cause the number of configurations to be greater than what would be valid for the given code. This does not affect the correctness since only the generated code corresponding to the right configurations will get used—the extra configurations simply represent wasted compiler effort. In the case of the DSP procedures studied it turns out that if argument types are pinned down through annotations on the argument variables then exactly one type-configuration is valid.

Figure 6 shows the number of type-configurations generated for five different DSP procedures by the constraints-based inference algorithm. The left darker bars indicate the number of configurations generated without any annotations on the arguments. The lighter bars indicate the number of type-configurations generated when the arguments have been annotated with their precise types that are expected by the library writer.

The fact that the lighter bars are not all one (only the leftmost, for `acf`, is one) shows that the static constraints-based algorithm does have limitations that get translated to more than the necessary number of type-configurations. However, these numbers are not very large—all, except `fourier_by_jump`, are smaller than 10—showing that the static analysis performs reasonably well in most cases.

Another important observation here is that annotations on the libraries serve as a very important aid to the compiler. The substantial difference in the precision of the algorithm with and without annotations indicates that the hints from the library writer can go a long way in nudging the compiler in the right direction. This conclusion also validates the strategy of making library writers' annotations an important part of the library compilation process in the telescoping languages approach.

4.2 Effectiveness of Slice-hoisting

Having verified that there is a need to plug the hole left by the limitations in the constraints-based inference algorithm, we conducted another set of experiments on the same procedures to evaluate the effectiveness of slice-hoisting. Figure 7 shows the percentages of the total number of variables that are inferred by various mechanisms. In each case, exactly one type-configuration is produced, which is the only valid configuration once argument types have been determined through library-writer's annotations. In one case, of `acf`, all the arguments can be inferred without the need for any annotations. The results clearly show that for the evaluated procedures slice-hoisting successfully inferred all the array-sizes that were not handled by the static analysis.

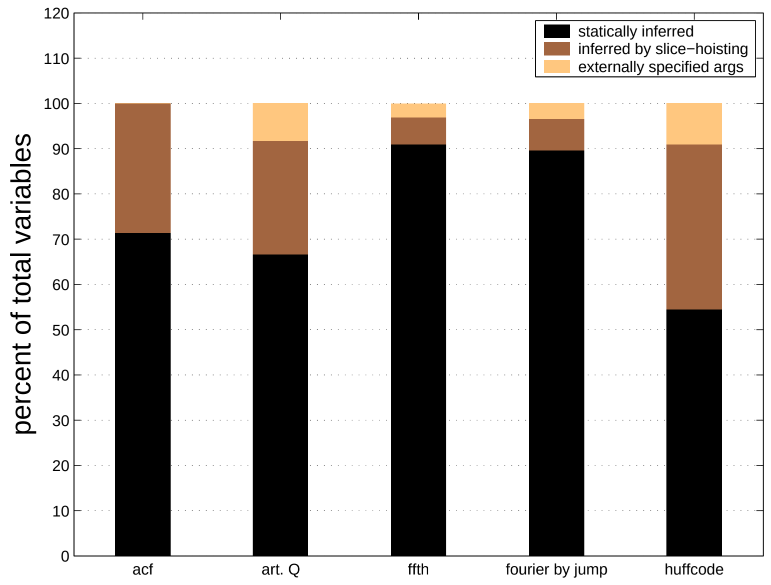


Fig. 7. Value of slice-hoisting

4.3 Implementation

The constraints-based static type inference has been implemented as a type-inference engine in the library compiler for MATLAB that is being developed at Rice as a part of a telescoping languages compiler. The various number of configurations shown in Fig. 6 are based on this engine. Slice-hoisting is under implementation and the number of variables shown to be inferred through slice-hoisting in Fig. 7 are based on a hand-simulation of the slice-hoisting algorithm. The implementation is expected to be ready by the time of the workshop.

5 Related Work

Conceptually, slice-hoisting is closely related to the idea of inspector-executor style pioneered in the Chaos project at the University of Maryland, College Park by Saltz [13]. That style was used to replicate loops to perform array index calculations for irregular applications in order to improve the performance of the computation loops. In certain cases, hoisted slices can reduce to an inspector-executor style computation of array sizes to avoid the cost of array resizing in loops. However, the idea of slice-hoisting applies in a very different context and is used to handle a much wider set of situations.

Type inference for MATLAB was carried out in the FALCON project at the University of Illinois, Urbana-Champaign [7, 6]. A simplified version of FALCON's type inference was later used in the MaJIC compiler [1]. The FALCON compiler uses a strategy based on dataflow analysis to infer MATLAB variable types. To perform array-size inference that strategy relies on shadow variables to track array sizes dynamically. In order to minimize the dynamic reallocation overheads it uses a complicated symbolic analysis algorithm to propagate symbolic values of array-sizes [14]. Slice-hoisting, on the other hand, can achieve similar goals through a much simpler use-def analysis. Moreover, if an advanced symbolic or dependence analysis is available in the compiler then it can be used to make slice-hoisting more effective. Finally, even very advanced symbolic analysis might not be able to determine sizes that depend on complicated loops while slice-hoisting can handle such cases by converting them to the inspector-executor style.

An issue related to inferring array-sizes is that of storage management. Joisha and Banerjee developed a static algorithm, based on the classic register allocation algorithm, to minimize the footprint of a MATLAB application by reusing memory [8]. Reducing an application's footprint can improve the performance by making better use of the cache. If a hoisted slice must be executed at runtime to compute the size of an array then the array will be allocated on the heap by Joisha and Banerjee's algorithm. Their algorithm can work independently of—and even complement—slice-hoisting.

Type inference, in general, is a topic that has been researched well in the programming languages community, especially in the context of functional programming languages. However, inferring array sizes in weakly typed or untyped languages is undecidable in general and difficult to solve in practice. Some attempts have been made at inferring array sizes by utilizing dependent types in the language theory community. One such example is eliminating array-bound checking [15].

6 Conclusion

Type-inference is an important step in compiling MATLAB. Precise type information can greatly improve the generated code resulting in substantial performance improvement. Inferring array-sizes turns out to be a difficult problem, while not having precise array-size information can lead to very expensive array-copy operations at runtime.

This paper has presented a new technique to perform array-size inference that complements the constraints-based static type-inference approach. The technique, called slice-hoisting, relies on very simple code transformations without requiring advanced analyses. At the same time, availability of advanced analyses can improve slice-hoisting either by making it possible to hoist slices where it might have been deemed undoable due to imprecise dependence information, or by improving the static evaluation of the hoisted slice.

Evaluation of the technique on a selection of DSP procedures has demonstrated its effectiveness in plugging the holes that are left by a purely static constraints-based approach to infer array-sizes.

7 Acknowledgments

We thank Randy Allen for making some of the MATLAB procedures available from the MathWorks web-site in a ready-to-use form. Thanks to Vinay Ribeiro, Justin Romberg, and Ramesh Neelamani for making their MATLAB code available for our study and to Behnaam Aazhang who heads the Center for Multimedia Applications that has an ongoing collaboration with the telescoping languages effort.

References

1. George Almási and David 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.
2. Arun Chauhan and Ken Kennedy. Procedure strength reduction and procedure vectorization: Optimization strategies for telescoping languages. In *Proceedings of ACM-SIGARCH International Conference on Supercomputing*, June 2001.
3. Arun Chauhan and Ken Kennedy. Reducing and vectorizing procedures for telescoping languages. *International Journal of Parallel Programming*, 30(4):289–313, August 2002.
4. Arun Chauhan, Cheryl McCosh, Ken Kennedy, and Richard Hanson. Automatic type-driven library generation for telescoping languages. To appear in the Proceedings of SC: High Performance Networking and Computing Conference, 2003.
5. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.
6. Luiz DeRose and David Padua. Techniques for the translation of MATLAB programs into Fortran 90. *ACM Transactions on Programming Languages and Systems*, 21(2):286–323, March 1999.
7. Luiz Antônio DeRose. *Compiler Techniques for Matlab Programs.* PhD thesis, University of Illinois at Urbana-Champaign, 1996.
8. Pramod G. Joisha and Prithviraj Banerjee. Static array storage optimization in MATLAB. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2003.
9. Ken Kennedy, Bradley Broom, Keith Cooper, Jack Dongarra, Rob Fowler, Dennis Gannon, Lennart Johnson, John Mellor-Crummey, and Linda 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, December 2001.
10. Kathleen Knobe and Vivek Sarkar. Array SSA form and its use in parallelization. In *25th Proceedings of ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, January 1998.
11. Cheryl McCosh. Type-based specialization in a telescoping compiler for MATLAB. Master's thesis, Rice University, Houston, Texas, 2002.
12. Vijay Menon and Keshav Pingali. A case for source level transformations in MATLAB. In *Proceedings of the ACM SIGPLAN / USENIX Conference on Domain Specific Languages*, 1999.

13. Shamik Sharma, Ravi Ponnusamy, Bongki Moon, Yuan-Shin Hwang, Raja Das, and Joel Saltz. Run-time and compile-time support for adaptive irregular problems. In *Proceedings of SC: High Performance Networking and Computing Conference*, November 1994.
14. Peng Tu and David A. Padua. Gated SSA-based demand-driven symbolic analysis for parallelizing compilers. In *Proceedings of ACM-SIGARCH International Conference on Supercomputing*, pages 414–423, 1995.
15. Hongwei Xi and Frank 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.