

RICE UNIVERSITY  
**A Type-Based Prototype Compiler for Telescoping  
Languages**  
by  
**Cheryl McCosh**

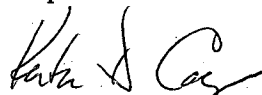
A THESIS SUBMITTED  
IN PARTIAL FULFILLMENT OF THE  
REQUIREMENTS FOR THE DEGREE  
**Doctor of Philosophy**

APPROVED, THESIS COMMITTEE:

Deceased

---

Ken Kennedy, Professor, Advisor  
Computer Science



---

Keith Cooper, Professor, Chair  
Computer Science



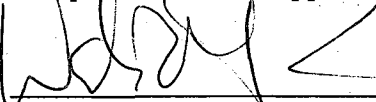
---

John Mellor-Crummey, Professor  
Computer Science



---

Dan Sorensen, Professor  
Computational and Applied Mathematics



---

Walid Taha, Assistant Professor  
Computer Science

Houston, Texas  
December, 2008

UMI Number: 3362354

### INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

**UMI<sup>®</sup>**

---

UMI Microform 3362354  
Copyright 2009 by ProQuest LLC  
All rights reserved. This microform edition is protected against  
unauthorized copying under Title 17, United States Code.

---

ProQuest LLC  
789 East Eisenhower Parkway  
P.O. Box 1346  
Ann Arbor, MI 48106-1346

# A Type-Based Prototype Compiler for Telescoping Languages

Cheryl McCosh

## Abstract

Scientists want to encode their applications in domain languages with high-level operators that reflect the way they conceptualize computations in their domains. Telescoping languages calls for automatically generating optimizing compilers for these languages by pre-compiling the underlying libraries that define them to generate multiple variants optimized for use in different possible contexts, including different argument types. The resulting compiler replaces calls to the high-level constructs with calls to the optimized variants. This approach aims to automatically derive high-performance executables from programs written in high-level domain-specific languages.

TeleGen is a prototype telescoping-languages compiler that performs type-based specializations. For the purposes of this dissertation, types include any set of variable properties such as intrinsic type, size and array sparsity pattern. Type inference and specialization are cornerstones of the telescoping-languages strategy. Because optimization of library routines must occur before their full calling contexts are available, type inference gives critical information needed to determine which specialized variants to generate as well as how to best optimize each variant to achieve the highest performance. To build the prototype compiler, we developed a precise type-inference algorithm that infers all legal type tuples, or type configurations, for the program variables, including routine arguments, for all legal calling contexts.

We use the type information inferred by our algorithm to drive specialization and

optimization. We demonstrate the practical value of our type-inference algorithm and the type-based specialization strategy in TeleGen.

## Acknowledgments

I would first like to thank my late advisor, Ken Kennedy, for his constant support and faith in me. He will remain one of the biggest influences in my life, and I will always remember the example he set both as a researcher and as a human. I cannot say enough about what he has meant to me or how much he is missed.

I would also like to thank my Chair, Keith Cooper, who has acted as my advisor for the last couple of years. His support and patience have ultimately pulled me through.

I would like to thank my committee members. John Mellor-Crummey has given me many insights and much advice. I appreciate his help and interest in my my research, especially in the last couple of years. Dan Sorensen helped me with ARPACK and the MATLAB codes. Walid Taha spent a lot of time helping me both in writing and in formalizing my work, as well as helping me understand some of the history of programming languages.

A number of agencies have supported me financially throughout my tenure at Rice. This material is based on work supported by the National Science Foundation and the Defense Advanced Research Projects Agency under Grant No. NSF CCF-0444465, the Department of Energy from the Los Alamos National Laboratory, the State of Texas Advanced Technology Program (ATP) under Grant No. 003604-0061-2001, the Department of Defense, contract No. H98230-04-C-0455, the Department of Energy Office of Science, Cooperative Agreement No. DE-FC02-07ER25800, and National Instruments under Agreement No. SRA-05012602.

I would like to thank Eric Allen for the internship opportunity at Sun. I learned many valuable lessons in my time there. Eric was also instrumental in helping me to

understand some of the programming-languages constructs. He was instrumental in helping me formalize my type-inference algorithm.

There have been many people at Rice that have supported me both with my thesis work and in helping me get through the tough times, of which there have been many. I would especially like to thank Tim Harvey who put up with my almost daily visits and for caring that I actually finish. Without Tim, I do not know if I would have gotten through the last few years. Arun Chauhan, Jason Eckhardt, and Mary Fletcher all greatly contributed to the design and implementation of TeleGen. Penny Anderson, Mike Fagan, Ronald Garcia, John Garvin, Richard Hanson, Guohua Jin, Mack Joyner, Chuck Koelbel, Gabriel Marin, Samah Abu-Mahmeed, Todd Waterman, Edwin Westbrook, Qing Yi, Rui Zhang, and Fengmei Zhao supported me in multiple capacities.

Finally, I would like to thank my family. I would like to especially thank my mom for coming from Georgia several times over the last couple of years to take care of my daughter so I could focus on my research and my dad for doing without her during those times. I would like to thank my daughter, Maeve, for bringing joy and a whole new purpose to my life, for not getting sick the last couple of months before my defense, and for putting up with my lack of complete focus on her with amazing patience. Most importantly, I would like to thank my husband who has supported me almost tirelessly through this whole process and who took on the lion's share of housework and child rearing in addition to a more than full-time job as I tried to finish. I have been truly blessed in my home life and in my time at Rice.

# Contents

Abstract	ii
List of Illustrations	x
<b>1 Introduction</b>	<b>1</b>
1.1 Telescoping Languages . . . . .	2
1.2 Types and Telescoping Languages . . . . .	4
1.3 TeleGen . . . . .	6
1.4 ARPACK . . . . .	9
1.5 Thesis . . . . .	9
1.6 Organization . . . . .	10
<b>2 Related Work</b>	<b>12</b>
2.1 Projects Related to Telescoping Languages . . . . .	13
2.1.1 Development of Domain-Specific Languages . . . . .	13
2.1.2 Library Compilation . . . . .	14
2.1.3 Component Integration . . . . .	16
2.2 Type Inference . . . . .	16
2.2.1 Hindley-Milner Type Inference . . . . .	17
2.2.2 Subtyping with Union and Intersection Types . . . . .	18
2.2.3 Dependent Types and Componential Set-Based Analysis . . . . .	19
2.2.4 Type Inference for Object-Oriented Languages . . . . .	19
2.2.5 Type Inference Using Data-Flow Analysis . . . . .	20
2.3 Specialization . . . . .	21
2.4 Compiling MATLAB . . . . .	22

2.4.1	Mathworks compilers . . . . .	23
2.4.2	FALCON . . . . .	23
2.4.3	MaJIC . . . . .	25
2.4.4	MAGICA and MATCH . . . . .	26
2.4.5	Menhir . . . . .	27
2.4.6	Type Inference via Partial Evaluation . . . . .	28
2.4.7	Cost of Interpretation . . . . .	28
2.4.8	Other Parallel MATLAB Compilers . . . . .	28
2.4.9	MATLAB-Like Languages . . . . .	28
<b>3</b>	<b>Type Inference</b> . . . . .	<b>30</b>
3.1	Motivating Type Inference . . . . .	30
3.2	The Type-Inference Problem . . . . .	32
3.2.1	Type Problems . . . . .	33
3.2.2	Type Inference in MATLAB . . . . .	34
3.2.3	Type Inference in Telescoping Languages . . . . .	35
3.2.4	Existing Solutions . . . . .	36
3.3	Type-Inference Solution for Size Problem . . . . .	37
3.3.1	Statement Information . . . . .	38
3.3.2	Combining Statement Information . . . . .	40
3.4	An Implementation for Solving Procedure Constraints . . . . .	42
3.4.1	Reducing Type Inference to Clique Finding . . . . .	43
3.4.2	Finding Cliques . . . . .	45
3.4.3	Solving Cliques . . . . .	48
3.5	Formal Description . . . . .	52
3.5.1	CORE-MATLAB . . . . .	52
3.5.2	Deriving the Principal Type . . . . .	61



<b>4</b>	<b>Extensions to Type-Inference Algorithm</b>	<b>64</b>
4.1	Handling Control Flow . . . . .	64
4.1.1	Original Strategy . . . . .	65
4.1.2	Complete Strategy . . . . .	66
4.1.3	Complexity . . . . .	69
4.1.4	Specialization . . . . .	69
4.2	Subscripted Array Accesses . . . . .	70
4.3	Value Dependent Operators . . . . .	72
4.4	Using Annotations . . . . .	73
4.5	Interprocedural Type Inference . . . . .	74
4.5.1	Function Parameters . . . . .	74
4.5.2	Missing Return Type-Jump-Functions . . . . .	75
4.5.3	Recursion . . . . .	75
4.6	Other Type Problems . . . . .	77
4.6.1	Handling Multiple Dimensions . . . . .	78
4.6.2	Pattern and Intrinsic Type Problems . . . . .	78
4.7	Distribution Inference . . . . .	80
4.8	Type Inference for Object-Oriented Languages . . . . .	81
4.8.1	Type Lattice . . . . .	82
4.8.2	Extending the Lattice . . . . .	84
<b>5</b>	<b>Type-Based Specialization</b>	<b>87</b>
5.1	A Simple Model for Specialization . . . . .	87
5.2	Variant Generation . . . . .	88
5.2.1	Beneficial Variants . . . . .	89
5.2.2	Most Used Variants . . . . .	92
5.3	Variant Selection . . . . .	93
5.3.1	Selection from Output Types. . . . .	95

5.3.2	Coercion Problem . . . . .	95
<b>6</b>	<b>TeleGen</b>	<b>97</b>
6.1	Structure of TeleGen . . . . .	97
6.1.1	Type Inference in TeleGen . . . . .	98
6.1.2	Type-Based Specialization . . . . .	98
6.1.3	Code Generation . . . . .	99
6.1.4	Limitations to Current Implementation of the Compiler . . . . .	99
6.2	Experimental Evaluation . . . . .	101
6.2.1	Experimental Setup . . . . .	101
6.2.2	Evaluation of Type-Inference Algorithm . . . . .	103
6.2.3	Code Generation . . . . .	109
6.2.4	Value of Specialization . . . . .	110
6.2.5	Library Generation . . . . .	111
<b>7</b>	<b>Contributions and Future Work</b>	<b>113</b>
7.1	Contributions . . . . .	114
7.2	Future Work . . . . .	115
<b>A</b>		<b>118</b>
A.1	Error Propagation Rules and Type Safety . . . . .	118
A.2	Normalization and Type Preservation . . . . .	121
A.3	Soundness . . . . .	123
A.4	Completeness . . . . .	124
	<b>Bibliography</b>	<b>126</b>

# Illustrations

1.1	Overview of the telescoping-languages approach. . . . .	3
1.2	Type-based specialization in a telescoping compiler. . . . .	8
3.1	Example of a procedure type for the size inference problem on the MATLAB “*” operation. Each clause gives a possible size configuration over the sizes of the variable. The first clause states that the operation could be scalar multiplication. This second and third clauses state that “*” could be a scaling operation. The fourth clause shows the operation could be matrix-matrix or matrix-vector multiplication. The last clause gives the possibility that the operation is multiplication of two vectors. This last case is necessary to keep track of the fact that $o1$ may be scalar when $i1$ and $i2$ are not. While the last clause may not represent different functionality from the fourth clause for this “*” operation, the output may be used in another statement, where understanding that it is scalar may be important to determining the meaning of the operation. . . . .	39
3.2	The resulting type configurations and the corresponding pruned SSA form of ArnoldiC. . . . .	41
3.3	Example graph. . . . .	44
3.4	Iterative n-clique finding algorithm. . . . .	47
3.5	CORE-MATLAB syntax. . . . .	52
3.6	CORE-MATLAB operational semantics. . . . .	55
3.7	Type language. . . . .	57

- 3.8 Type system for size problem. . . . . 58
- 3.9 Simple constraint system for size problem. . . . . 62
- 3.10 Well-formed type configurations. . . . . 63
  
- 4.1 Type inference in the presence of control flow. . . . . 67
- 4.2 Statement constraints in the presence of array sections. . . . . 70
- 4.3 Return type-jump-function for MATLAB operators `size` and `zeros`. . . 72
- 4.4 Interprocedural algorithm. . . . . 76
- 4.5 Intrinsic type constraints on mean operation. . . . . 79
- 4.6 Example classes written in Python. . . . . 82
- 4.7 Type Lattice for Python with user-defined types. . . . . 83
- 4.8 Return type-jump-function for `move` and `area`. . . . . 83
- 4.9 Concise return type-jump-function for `move`. . . . . 84
- 4.10 Example type extension. . . . . 85
  
- 6.1 Results of type inference. . . . . 103
- 6.2 Results of size inference. . . . . 103
- 6.3 Results of intrinsic-type inference. . . . . 104
- 6.4 Results of shape inference. . . . . 104
- 6.5 Comparison of MATLAB performance versus automatically generated  
Fortran. . . . . 108
- 6.6 Value of Specialization. . . . . 110
- 6.7 Comparison to hand-coded Fortran. . . . . 111
  
- A.1 Error Propagation Rules . . . . . 119
- A.2 Normalization Rules. . . . . 121

# Chapter 1

## Introduction

A serious hindrance to the productivity of the scientific computing community is the time it takes to produce high-performance applications. This is due in part to the increasing complexity of modern-day architectures and to the lack of expert programmers. Scientists would like to abstract away from details of tuning for modern architectures and focus only on the high-level concepts of their domains. To increase their productivity, scientists have been looking to high-level, domain-specific languages to encode their applications.

Unfortunately, these languages have traditionally not been able to exploit key architectural features (*e.g.*, memory hierarchy) to achieve the highest possible performance. Therefore, scientists only use these languages to develop and test their algorithms. To achieve high performance for larger problems, they must hand-translate the high-level code into lower-level languages such as Fortran or C. This translation step consumes a large portion of development time.

Telescoping languages is a strategy that aims to decrease the time to develop high-performance, scientific applications by enabling scientists to encode applications in languages with high-level, polymorphic constructs that capture the domain functionality, while also delivering high performance for their applications. It accomplishes this by automatically generating optimizing compilers for domain-specific languages specified as libraries.

We built a prototype telescoping language compiler, TeleGen that focuses on type-based specialization, which is an integral part of any telescoping-languages system.

## 1.1 Telescoping Languages

Many scientists would benefit from the ability to encode applications in high-level, polymorphic languages. Such an approach enables programs to be expressed concisely with a minimum of effort. Additionally, having domain-specific functionality encoded directly in the language would make it easier for them to focus on only the problem at hand. Many higher-level languages are augmented with domain-specific toolboxes that capture such functionality. These toolboxes are typically encoded as libraries and extend the language in which prototype codes for scientific applications are written.

Using libraries can cause performance problems for applications. Libraries are typically treated as black-boxes, which is an impediment to applying optimizations once their calling contexts are known. Another option would be to recompile the library procedures used directly and indirectly by the application each time the application is compiled. This approach can achieve higher performance from the libraries at the cost of long compilation times, since compilation of the application becomes proportional to the size of the call chain through the libraries rather than the size of the application. Neither strategy exploits the domain-specific knowledge encoded in the libraries.

A more powerful solution would call for application developers to use a domain-specific language with its own optimizing compiler. This solution offers the advantage that the domain-specific knowledge can be directly encoded in the compiler. The problem with this strategy is that the time required to develop languages and compilers is prohibitive, especially given that the size of the user base is typically small. For most domains, this strategy is simply not an option.

We need a strategy that not only makes it simple to develop and maintain new domain-specific languages, but can also exploit the inherent algebra in the domains to achieve high performance. Telescoping languages is a strategy that seeks to simplify application development of high-performance implementations by automatically deriving them from high-level specifications based on programs built using domain

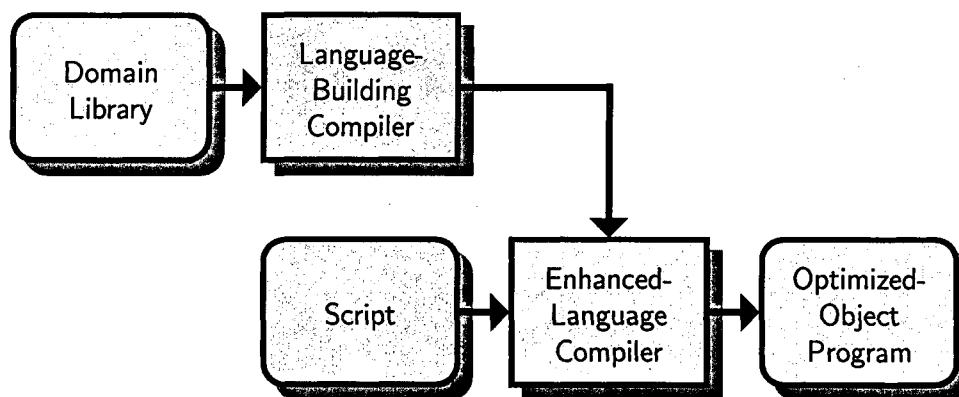


Figure 1.1 : Overview of the telescoping-languages approach.

libraries.

In the telescoping-languages strategy, primitive operations in domain-specific languages are encoded as libraries. Encoding languages as libraries also has the advantage of ease of development. “Languages” can be built as extensions to languages that have already been generated by the telescoping compiler. The idea of developing applications from multi-layered libraries led to the term *telescoping languages*.

Telescoping languages envisions automatically building the domain-specific language through the *language-building compiler*. User scripts, or the applications encoded in these languages, can then be compiled using the language compiler to produce highly-optimized object code. Note that scripts can be encoded in any language, since the scripts merely assemble together calls to the libraries. The telescoping-languages strategy is shown pictorially in Figure 1.1.

To transform one or more libraries into a domain-specific language, the telescoping-languages strategy involves an extensive pre-compilation phase of the library. During this phase, analysis of the library procedures is performed to determine the possible contexts in which each procedure is used. This analysis is guided by annotations provided by the library writer that specify the algebra for the library.

These annotations are based on the library annotations used in Broadway [45, 46, 44]. However, Broadway does not pre-compile the libraries. In this way, a telescoping-languages system is able to exploit domain-specific knowledge.

After analysis, the library compiler generates variants optimized for several possible uses of each library procedure. The enhanced-language compiler then merely needs to replace sequences of calls to the library procedures with calls to the appropriate variants. As a result, compilation time for user scripts remains linear in the size of the scripts.

Because the responsibility of achieving high performance for the multiple possible uses of the library is transferred to the library generator, library developers can also benefit from the telescoping-languages strategy. They can encode their libraries in a single, polymorphic version written in a high-level, domain-specific language, and the language generator will ensure that the library is optimized for every use that may occur in practice. Note that generating variants for libraries is similar to performing script compilation since libraries can script together calls to previously generated libraries.

Library generation is a costly step, primarily due to the fact that aggressive optimizations are performed for multiple possible uses of the library. However, the time to perform library generation can be amortized over the many uses of the library, since it should only need to be performed infrequently for maintenance purposes. Also, extra compilation time during library generation will still be less time consuming than the hand-translation step typically performed by library developers.

## 1.2 Types and Telescoping Languages

Type inference and specialization are cornerstones of the telescoping-languages strategy. Since the telescoping-languages strategy envisions generating multiple specialized versions of library procedures and recognizing which version is most appropriate for a given call site, properties of the procedure arguments must drive the specializa-



tion choices. A vast majority of these properties can be viewed as types, which is a term we use for the purposes of this dissertation in the loosest sense. Therefore, a telescoping-languages compiler must rely heavily on type inference to know which variants to generate in the libraries as well as which variants to call in the scripts.

To build a compiler for telescoping languages, we needed a precise type-inference algorithm. Type inference should limit the number of variants generated to only those that could occur without error in practice. Because libraries are pre-processed without knowledge of the calling context, type inference must be able to infer the types of the procedure arguments from the uses in the procedure body. Type inference must work with information provided by the library writer, either in the form of annotations or sample scripts to enhance the type information inferred.

Also, since type inference will be used to infer properties that have non-traditional behavior with respect to the traditional notion of types and must infer types for scripting languages that are potentially not strongly typed, type inference needs to be more flexible than the type-inference schemes available from the programming-languages community. For example, the transpose of an upper triangular matrix is a lower triangular matrix and vice versa. While these kinds of type assignments are rare, they are valid and therefore need to be covered in the generated variants. Note that simply inferring that the input and output can both either be upper or lower triangular is not precise enough and would require generating four variants as opposed to only the two that are valid.

There were no existing type-inference strategies that could handle all of the features of telescoping languages. To solve these problems, we developed a novel type-inference strategy that can infer all possible type assignments, or type configurations, for the variables, including all possible calling contexts in terms of types. Types are inferred across procedures by summarizing the calling contexts inferred into a table that can be accessed by library procedures or scripts.

We use the type information inferred by our algorithm to drive specialization,

which includes both variant generation (choosing which specialized variants to generate) and variant selection (choosing the appropriate variants to call in place of calls to the generic library routines). We incorporated our type-inference algorithm and the type-based specialization strategy in a prototype compiler to demonstrate the practical value of these strategies.

### 1.3 TeleGen

TeleGen is a prototype compiler for telescoping languages that only includes the analysis and tools necessary to perform type-based specialization for libraries. This version of TeleGen will provide a building block for future telescoping-languages technologies, including new analyses and specializations.

The prototype compiler currently uses MATLAB as its base language. MATLAB is an array-based language for numerical computation and includes several toolboxes for a number of problem domains. MATLAB is widely-used within the scientific community precisely because it abstracts away low-level details. Unfortunately, MATLAB is unable to achieve the high performance needed by the scientific applications, leading many developers to recode their applications in lower-level languages such as Fortran or C. This translation step can take up a large portion of application development time. The telescoping-languages strategy envisions automating this last step in the application development process. MATLAB is an ideal starting point from which to test the ideas and new technologies in telescoping languages.

MATLAB is an interpreted language, and a number of issues arise when converting it to a compiled language. The most difficult of these is that MATLAB is a dynamically typed language with operators that are heavily overloaded based on type. Here, “type” refers to array properties such as array rank, size, sparsity pattern, and intrinsic type (integer, real, etc.). Therefore, to translate into a lower-level, explicitly-typed language and correctly determine the meaning of the operations, the types must be inferred by the compiler. MATLAB introduces interesting challenges for type infer-

ence. Even with the whole program available, statically inferring a single type for each variable in MATLAB is impossible in the general case.

Once analysis has been performed on the library procedures or scripts, code generation and specialization can commence.

Figure 1.2 shows the interactions between type inference and specialization in TeleGen. The pieces shown are the minimal pieces required to perform type-based specialization for high performance. Further analyses and optimizations could increase the performance achieved. To describe the pieces of TeleGen, we will not distinguish between procedure calls and operations. This is consistent with the telescoping-languages strategy since one of its goals is for procedure calls to behave as primitive operations in a higher-level language.

**Type Jump-Functions** The TeleGen compiler has the burden of inferring types when specific calls to a procedure are not given. Type inference must result in types defined in terms of the inputs so that, for each possible configuration of input types, a variant is generated with the correct corresponding types for the variables local to each procedure. To handle this, *type jump-functions* akin to those used in interprocedural analysis are defined [15, 43]. We use a tabular representation of the type jump-function so that each entry represents one possible type configuration. The type jump-function is only used for specialization. Once the procedure has been specialized, the table can be deleted.

*Return type-jump-functions*, which define the types of the outputs in terms of the types of the inputs, handle the propagation of type information across procedure calls.

**Code Generation** In the simplest model for specialization, one variant is generated for each entry in the type jump-function table. Each variant is specialized with respect to the types given in the entry by replacing each operation or procedure call within the procedure with a call to a variant that is specialized for the given types.

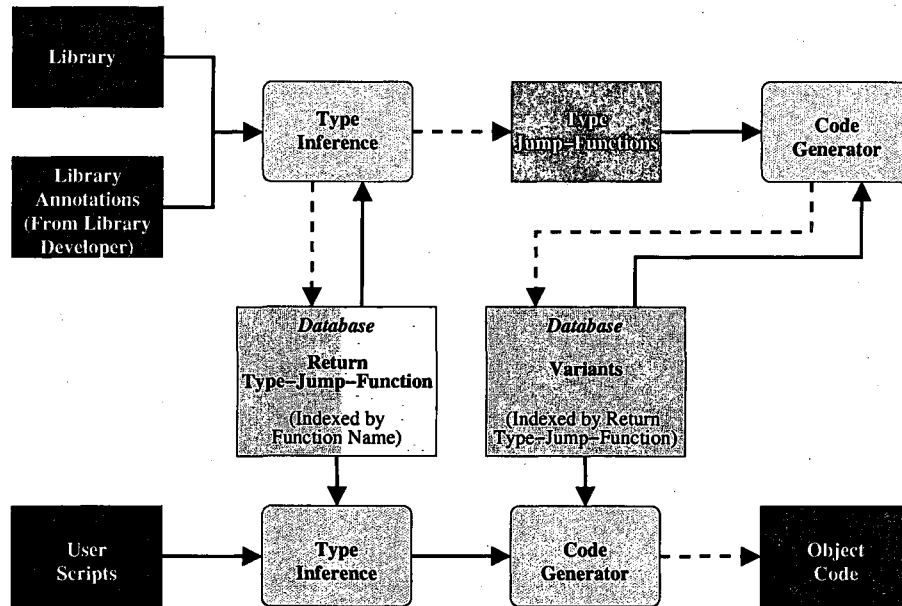


Figure 1.2 : Type-based specialization in a telescoping compiler.

**Interprocedural Analysis** To ensure this, TeleGen performs type inference on the library procedures in reverse post-order on the partial call graph, thus ensuring where possible that the return type-jump-functions are available for called procedures. If there is a cycle in the call graph, TeleGen iterates over the cycle until a fixed point is reached. There exists a fixed point since the compiler initializes the types to be the most general (*i.e.*, all type configurations over the arguments) and only reduces the number of type configurations at each iteration over the cycle. When source code is unavailable, as in the case of MATLAB's primitive operators, return type-jump-functions may be entered by hand.

**Putting It Together** Figure 1.2 describes how these pieces work together in a telescoping compiler. For library generation, the input to the type-inference engine is a library procedure and optional annotations that describe its possible uses. The type-inference engine uses information about called procedures and operations,

summarized in the return type-jump-function table, to determine the legal types of variables at every point in the procedure. This information is used to produce both a type jump-function and a return type-jump-function for the procedure. Code is generated using the type jump-function as well as a table of previously generated, specialized variants of called procedures.

Script compilation is similar to library generation. However, information does not need to be stored back to the tables since the scripts are only used for a single application.

## 1.4 ARPACK

Throughout this dissertation, we use examples from ARPACK, a linear algebra library for solving large-scale eigenvalue systems [74]. The ARPACK developers first developed their code in MATLAB. They then spent a large amount of time hand-translating the algorithm efficient Fortran 77 code. There are actually eight separate variants of the Fortran 77 code specialized to different types of the input matrix. This makes ARPACK an ideal example of an application for which telescoping-languages technology would be highly beneficial, especially type-based specialization. The primary example routine from the MATLAB version of the code is `ArnoldiC`.

## 1.5 Thesis

The ability to write high-level, polymorphic code improves productivity. Library-based scripting languages provide abstract operations that can be used to rapidly assemble complex applications. However, this approach can result in lower performance compared to application encoded in lower-level languages due to interpretive overhead.

This thesis explores strategies for type-based specialization in library-based programming models. The strategy is inherently interprocedural and focuses on making it possible to analyze and optimize multiple layers of libraries efficiently. Telescoping

languages aims to invest in offline analysis to be able to understand library operations in detail. It then uses the results of this analysis to generate a compiler that accepts calls to the library as primitive operators.

Type inference and specialization are cornerstones of the telescoping-languages effort. They are indispensable elements of a system that analyzes and optimizes multiple layers of library-based primitives.

This document shows that type-based specialization is important to achieve high performance and that we are able to achieve effective specialization with a manageable number of variants. We describe type inference in this system and show that it is sufficient to solve types in the library generation problem. Type inference discovers a set of type configurations that cover the set of possibilities that could occur for the library subroutines. We carefully analyze its algorithmic complexity to both identify the worst case and to show that type inference is efficient in practice. We also show that the result of type inference is precise, thereby limiting the number of generated variants to only those that are valid (in terms of types) in practice. We support this analysis with experimental data. To provide concrete validation of the practical value of this strategy, we built a prototype compiler that uses the type analysis and the simple specialization technique described in this document.

## 1.6 Organization

Chapter 2 provides an overview of related work. We will then discuss the type-inference algorithm and formalize it to prove soundness and completeness in Chapter 3. In Chapter 4, we describe extensions to the type-inference algorithm that are necessary for type inference to work over MATLAB code as it is used in practice as well as extensions that enable a broader range of types to be inferred for other problems and languages. Chapter 5 describes the interactions between type inference and specialization. We describe our implementation of TeleGen in more detail in Chapter 6 and validate our strategy experimentally using TeleGen. Finally, in Chapter 7, we

discuss contributions and future work.

## Chapter 2

### Related Work

Van Deursen et. al. [77] define a domain-specific language to be “a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.” Some of the benefits to domain-specific languages that they mention include enabling solutions to be expressed at a level that can be understood and developed by domain experts, enhanced productivity, reliability and maintainability. Some drawbacks they mention are the cost of developing the languages and potential losses of efficiency inherent in the language.

Telescoping languages aims to provide users the benefits of high-level, domain-specific languages without requiring an extensive language-development process [56, 57]. Moreover, telescoping languages seeks to achieve high-performance from applications written in these languages.

There are many interesting areas of research within the telescoping-language project. Previous and current research in telescoping languages at Rice includes development of domain libraries with descriptions of possible high-level transformations [11], optimizing transformations for procedures called inside loops [17], and transformations that allow for earlier allocation of arrays [18, 19].

The focus of the work in this dissertation is type-based specialization, which is essential to generate high-performance code from scripting languages. This work also focuses on developing strategies to give the domain-library developers the ability to use high-level, domain-specific languages and to give the user the ability to encode their applications in these languages while still achieving high performance.



To solve these problems, we developed strategies for MATLAB compilation within the telescoping-languages framework, including a new strategy for type inference. These strategies build upon ideas developed in the author's Master's thesis [62].

The ideas and goals of telescoping languages are related to a number of different research projects. This chapter discusses some of those projects and describes work that is specifically related to the technologies developed in this dissertation.

## **2.1 Projects Related to Telescoping Languages**

The telescoping-languages strategy uses libraries to build domain-specific languages. There has been a lot of research on both developing domain-specific languages and library compilation. Telescoping languages combines these strategies.

### **2.1.1 Development of Domain-Specific Languages**

Developing a domain-specific language from scratch is impractical for most domains. Therefore, some projects have focused on developing domain-specific languages from existing languages. Two such projects are MAGIK and KHEPERA. Telescoping languages also aims to build domain-specific languages from existing languages, but uses libraries to define the extensions, which are simpler to use.

Engler developed a system called MAGIK, which provides mechanisms for programmers to include extensions to the compiler that can modify the system's intermediate representation during compilation [36]. Thus, high-level, domain-specific knowledge can be encoded into a general-purpose compiler. The developer must be very familiar with the intermediate representation and the compiler implementation.

KHEPERA provides a small language for specifying translations from a domain-specific language to another language with pre-existing high-performance compilers [64]. The translations must result in code that makes it easy for the target language's compiler to exploit optimization opportunities. In this way, the language developer can leverage existing compiler technology to achieve high performance.

This also requires the domain-specific language developer to have detailed knowledge about the target language's compiler.

### 2.1.2 Library Compilation

Several projects have looked into different strategies for library compilation. Libraries are widely recognized as a relatively simple way to encapsulate domain knowledge. Because they are traditionally treated as black boxes, they cannot get the performance required. The projects mentioned here focus on finding ways to make library usage more practical. The primary difference between these projects and telescoping languages is that telescoping languages seeks to turn library calls into primitive operators in a language.

The Broadway compiler developed at the University of Texas at Austin has very similar goals to those of telescoping languages in that Broadway allows domain-library writers to provide domain-specific annotations that aid the compiler in optimizing applications that use the library code [45, 44, 46]. Telescoping languages envisions using similar annotations to guide optimization in the language compiler. Broadway has shown that by using these annotations, the compiler can achieve performance improvements over compiling the application without domain-specific knowledge. A major difference between the Broadway strategy and the telescoping-languages strategy is that Broadway does not perform extensive pre-compilation of the libraries, but rather compiles the libraries together with the application. This can mean large compilation times even for small applications. Also, Broadway requires hand-written annotations for every layer of libraries, which is a laborious process. The type-inference strategy provided in this dissertation is in part designed to reduce the need for user annotations, although annotations will augment type inference.

The Rose project at the Lawrence Livermore National Laboratory is designed to provide a way for users to express high-level abstractions of library semantics in C++ using pattern-based rewrite rules [73]. This framework allows the user to express op-

timizations that can be used by the compiler when transforming the code. The user must be familiar with the representation of the AST to specify these transformations. There has also been work at Lawrence Livermore National Laboratory on making expression templates more efficient [9]. Expression templates allow expressions to be passed into the function as arguments and inlined into the function body [79]. This technique helps avoid unnecessary temporaries and enables more loop optimizations. The type inference and type-based specialization strategies presented here could benefit from similar constructs, since statements are flattened, causing many unnecessary temporaries.

Vandevorde developed support that allows programmers to define optimizations through the use of interfaces called SPIs or specialized procedure interfaces [78]. These interfaces allow the programmer to provide multiple implementations specified for different possible calling contexts. The compiler is responsible for replacing calls to the interface with calls to the appropriate implementation. To determine which implementation to use, the compiler uses conditions provided by the programmer and written in a formal-speculation language. The compiler must then prove whether the conditions hold. Telescoping languages seeks to automate the entire process. Also, theorem provers are not an ideal tool for compilation as they are computationally intensive. The telescoping-languages strategy also involves using computationally intensive tools during library generation, but this compilation phase occurs infrequently.

ATLAS and FFTW are examples of specialized libraries that are automatically tuned for specific platforms [81, 42]. The goals behind ATLAS and FFTW are complementary to those proposed by telescoping languages. Telescoping languages focuses on optimizing for different possible uses of the libraries while ATLAS optimizes for different architectures. In fact, we use ATLAS-tuned BLAS routines to handle the base library routines from which other libraries are built. In this way, the telescoping-languages compiler is also specialized for the different possible platforms.

SPIRAL is a library generator for high performance software implementations of

linear signal processing transforms [69]. It represents the transforms as formulas and then generates the algorithms that are optimized for the target architecture for the specified transforms. These algorithms are then translated into Fortran or C.

Active libraries are libraries that take an active role in compilation by generating components, specializing algorithms and configuring themselves for a target machine [80]. All of the projects mentioned in this subsection fall into the category of active libraries, including telescoping languages. Simply put, active libraries allow library developers to encode abstractions and corresponding optimizations into the libraries.

### **2.1.3 Component Integration**

Telescoping languages can also be viewed as a strategy for component integration, where the components are libraries [7]. The Common Component Architecture (CCA) effort's primary goal is to try to develop component integration systems for scientific application development. CCA trades performance for flexibility. Components are dynamically selected within CCA-compliant frameworks, and components are treated as black-boxes, prohibiting beneficial cross-component optimizations.

## **2.2 Type Inference**

The programming-languages community has focused primarily on type systems for the purpose of static type checking. These static type checkers are, in general, designed to ensure that a program will not compile unless no runtime type errors are possible. This may mean for some languages that well-typed programs will not pass the static type checker. In contrast, the type-inference system presented here is designed to identify optimization opportunities. The compiler must model the behavior of the language. Therefore, it needs to infer all valid type configurations of the variables, to cover any possible situation that may occur without a type error in the MATLAB interpreter.

Type reconstruction, or type inference, is the process of trying to infer the types of expressions that are not explicitly typed in the language [66]. In general, type reconstruction tries to determine principal types or most general types. Type information of this kind is not precise enough for optimization in a telescoping-languages compiler. Furthermore, most of the research in this area has been on type systems in functional languages with possibly some extensions to handle imperative constructs.

Most of these systems infer a traditional notion of type (intrinsic types or object types). However, a few provide type systems that can infer information such as array sizes, which we describe in Section 2.2.3.

### 2.2.1 Hindley-Milner Type Inference

Hindley-Milner type inference is a common technique for performing polymorphic type inference in functional programming languages [29]. Hindley-Milner determines a polymorphic type for each expression in a program independently. Limited dependence between types is expressed through the use of type variables. Hindley-Milner provides a basis for type inference with more complex type systems. Hindley-Milner is designed to infer principal types of expressions, where a principal type is a type from which all other valid types can be computed.

The Hindley-Milner type inference proceeds by placing typing constraints on expressions using type variables to represent type dependencies. These constraints are then solved using a unification process. While this sounds very much like the type-inference algorithm presented in this dissertation, the types of constraints and the fact that they are formed over expressions and not the variables makes the type information inferred by Hindley-Milner very different. Also, the style of overloading found in MATLAB makes it difficult for a Hindley-Milner-style type inference to infer types with the same precision.

Hindley-Milner type inference requires that the types of functions be uniquely determined given the types of the input parameters. This means that it is unable

to determine types of functions that could have different output types depending on the control flow within the function or the values of the parameters. These cases occur frequently in MATLAB functions. The following sections describe extensions to Hindley-Milner can handle this problem.

### 2.2.2 Subtyping with Union and Intersection Types

Subtyping provides constraints on the type system that allow functions to accept as inputs values that are “subtypes” of the parameter types (*i.e.*,  $b$  is a subtype of  $a$  means that the set of values of type  $a$  contain the set of values of type  $b$ ) [66]. This allows for code reuse in programs while maintaining the property of sound typechecking.

Union types handle the problem in Hindley-Milner, in which function types must be uniquely determined given the types of the parameters [75]. The result of a function may have a union of types representing the possibility that there were multiple paths within the procedure. Intersection types provide a way for the type system to allow finitary overloading [24, 25, 47, 67], which is sufficient for the intrinsic type and pattern problems, since these lattices are finite.

Subtyping with union and intersection types is able to express types that resemble the types inferred by our system. They are able to express the precise relationships between parameter types. Intersection types can more precisely type check programs, because of these more-exact relationships. However, type inference with intersection types is undecidable in the general case. Therefore, languages that use full intersection types require at least some explicit typing [71]. Many research projects have looked at ways to impose restrictions on the intersection types that would enable type inferring to be decidable [26, 16, 58, 41].

Very little work has been done on intersection types in imperative languages, making it more difficult to present a good comparison between intersection types and our work. One exception to this is work by Davies and Pfenning [30]. They show

that standard formulations of intersection type systems are unsound in imperative languages. They restrict intersection types to achieve soundness. Their formulation does require some explicit typing. Their system is designed for type checking as opposed to type reconstruction.

### 2.2.3 Dependent Types and Componential Set-Based Analysis

Dependent types are types that can be indexed by terms. To infer these dependent types, annotations from the programmer are required. Xi and Pfenning use dependent types to perform array bounds checks in ML programs [82]. Type inference in telescoping languages can infer array sizes in terms of the inputs without aid from the library developer, although such annotations can reduce both the run-time complexity and the space requirements for the compiled code.

Xi and Pfenning's approach does not allow for the overloaded operators present in MATLAB. For example, the `*` operator in MATLAB can be scalar multiplication, matrix scaling, or matrix multiplication, depending upon the shape of the parameters. Also, because they are inferring types to perform static type checking, their system accepts fewer valid programs than the size inference system presented here. Lastly, the information inferred from this system does not provide information for allocating the matrixes to the maximum size reached in any run of the program.

### 2.2.4 Type Inference for Object-Oriented Languages

Object-oriented languages pose special problems for type inference as well as many opportunities for highly beneficial optimizations. The primary complication in inferring types in such languages is that they are highly polymorphic in nature. Note that the systems mentioned below can also be used to infer types other than user-defined types, such as intrinsic types and patterns in MATLAB.

Agesen performed a study of the different constraint-based analyses for inferring types in languages with parametric polymorphism [2]. All of these systems rely on

whole-program analysis to precisely infer types. We present the systems that have the most in common with our type-inference algorithm here. In future work, we plan to show that combining types inferred for different components gives similar results to these whole-program analyses.

Palsberg and Schwartzbach developed a basic, fast constraint algorithm that assigns a type variable to every variable and expression in the program [65]. These type variables are nodes in the constraint graph. Edges or constraints provide data flow information from the program. Types are propagated to nodes through the constraints. The disadvantage of this scheme is that each method body is only represented in the graph once. If a method is invoked with different types, precision is lost.

Plevyak and Chien developed an improvement to this algorithm [68] that performs the basic algorithm and then uses the inferred type information to determine where methods should be cloned to improve precision. They do this iteratively until a fixed point is reached or until the algorithm reaches a fixed number of iterations. The types inferred are much more precise than with Palsberg and Schwartzbach's algorithm.

Agesen et. al. developed a cartesian product algorithm which is just as precise (in some cases more precise) than the algorithm by Plevyak and Chien [3]. An added bonus is that the algorithm is not iterative, and is therefore more efficient. The basic idea is that instead of cloning templates, cartesian products of all the possible parameter types are tracked.

### 2.2.5 Type Inference Using Data-Flow Analysis

Kaplan and Ullman developed a forward and backward data-flow scheme for type inference [55].<sup>1</sup> Forward and backward type-inference passes are performed to reduce the set of possible types for each variable until a fixed point is reached. This strategy requires that the type lattice have the finite-descending-chain property to prove termination. Therefore, it is not applicable to the problem of inferring array

---

<sup>1</sup>This algorithm reappears in a more accessible form in [4].



sizes, which we describe in Chapter 3. While this type-inference algorithm produces the most powerful results of any similar strategy, determining sets of possible types for the variables does not give precise enough information for library compilation in telescoping languages. It does not give exact relationships between the variables and their types. For example, one variable might be complex if and only if an input is complex. Yet, a type assignment that said that both could be either real or complex would suggest that a variant may be needed for the case when the input is real and the variable is complex.

### 2.3 Specialization

Specialization in the type-based prototype compiler presented in this dissertation involves variant generation and variant selection. Variant generation involves determining which of the inferred type configurations should induce a separate corresponding variant and which should (and can) be merged. Variant selection is the process of determining how to replace calls to the generic procedures with calls to the appropriate variants.

Variant generation is a form of procedure cloning [21, 22]. The methodology for determining when procedures should be cloned in these papers examines the call sites for the procedures. In the telescoping-languages strategy, not all of the call sites are available during library compilation. Therefore, the compiler must rely on precise analysis to determine which calling contexts could validly occur given the procedure body.

Dean, Chambers, and Grove developed a strategy for optimizing object-oriented languages by selectively cloning methods and specializing each for a possible receiver type [31]. The specialized versions have more information about the types of formal arguments, which means that methods called on these arguments may be statically dispatched, leading to further optimization opportunities. However, to determine which methods should be specialized, the compiler must examine the call graph to

see how the methods are called. This method will not work in telescoping languages, since it does not have the benefit of the full call graph when precompiling libraries.

Dean et al. extend the specialization methodology to specialize on all arguments, not just the receiver, which is also the aim of the type-based specialization described in this dissertation [32]. Their compiler uses profile information to reduce the amount of specialization, which cannot be applied to the current telescoping-languages framework without sample user scripts that provide the full range of types that may be seen. Instead, the library compiler must analyze the way in which arguments are used within the method to determine whether to specialize for a particular type. Excluding the possibility of increased space requirements, similar performance benefits could be achieved once the calling context is known.

## 2.4 Compiling MATLAB

MATLAB is a popular language within the scientific community for programming due to its simplicity, including the lack of explicit typing. However, scientific-application developers are not able to get the performance required for large-scale applications. Therefore, there have been several projects focused on moving MATLAB from a strictly interpreted language to a compiled language, where the compilers focus on achieving high performance.

MATLAB is a dynamically-typed language. Therefore, a single type for every variable is often not determinable at compile time. To compile MATLAB, types must be disambiguated. This is especially problematic since many of the operators in MATLAB are overloaded based on the types. All of the MATLAB compilation projects use some form of type inference to achieve higher performance as well as to maintain correctness. Type-inference strategies range from requiring that the whole-program be available to performing just-in-time compilation when the types can be known.

It is important to note that MATLAB compilation is not the primary focus of this dissertation. However, because we chose to use MATLAB as our source language,

TeleGen needs to handle all the problems inherent in compiling MATLAB. All of the technologies presented can be applied to other languages. However, MATLAB is a high-level language with a wide user-base and thus provides a good starting language for research in telescoping languages.

#### 2.4.1 Mathworks compilers

Mathworks provides a MATLAB-to-C compiler, called MATLAB Compiler (formerly MCC) [60, 59]. The compiler outputs either stand-alone applications or MEX files, which are C files that are built using an interface that can be called from the MATLAB interpreter. MATLAB Compiler's output at the top level seems to resemble the library calls that would be made in the interpreter. Every MATLAB operation has a corresponding library call that is passed the appropriate variables that are declared with the most general type - the array type. Most notably, the types of the variables are left undetermined in the top level and are interpreted further down the call chain to dispatch to the appropriate variants. This strategy precludes many simple optimizations, and the resulting code is comparable in performance to the original MATLAB.

#### 2.4.2 FALCON

FALCON, which was developed at the University of Illinois at Urbana Champaign by De Rose and Padua, is a MATLAB to Fortran 90 translator [72, 33]. Falcon takes as input a whole program written in MATLAB along with sample input data. The primary analysis needed for translation from MATLAB to Fortran is type analysis. The FALCON compiler infers intrinsic types, shape (scalar, vector, or matrix), and size that are included in the array properties inferred in TeleGen.

Static type inference is performed using data-flow analysis based on the algorithm developed by Kaplan and Ullman [55]. The analysis proceeds from the top-level procedure and sample data, and information is propagated forward through each

procedure. When a function call is encountered, its M-file is inlined into the top-level procedure. A table of MATLAB operators with type information that gives output types in terms of input types is used to infer types across the operators. This table is similar to the return-type-jump-function table in TeleGen, except that it only has summary information for the MATLAB operators. A single step of back propagation of type information occurs if the type of an array changes from an assignment. If the type of a variable is undeterminable through this process, it is given the type `unknown`.

Sizes that cannot be inferred statically from constants and `unknown` types are disambiguated at runtime. For intrinsic types, control-flow statements are entered that are predicated on the actual runtime type of the `unknown` variable. The branches of the statement include the type declaration of the variable and a cloned version of the operation that uses the variable. The dynamic sizes of matrix variables are also tracked, and tests of array sizes to check if the size is growing through assignment past the bounds of the original array are administered if needed after the static-size inference.

The FALCON compiler is able to achieve substantially better performance than the MATLAB interpreter and performance comparable to hand-coded Fortran 90 programs. While telescoping languages shares the goal of providing application developers the ability to produce high-performance code from high-level languages such as MATLAB, it also envisions reducing compilation costs by pre-compiling the underlying libraries and optimizing them for every possible use. Therefore, the need for the whole program to be available to the compiler, and the aggressive inlining strategy for both analysis and code generation employed by FALCON, are counter to the goals of telescoping languages. Also, inlining whole programs for type clarity may lead to significant source-code growth and subsequent high compilation costs.

Furthermore, not all types are inferred statically, since type inference in FALCON relies on data-flow analysis that converges to a single type. FALCON is able to avoid

these problems in most cases, since it has all the input-type information available during static analysis, and it has methods for coping with these variables dynamically. In contrast, the type-inference algorithm presented in this dissertation works over single procedures, without knowledge of the possible calling contexts. It infers multiple type configurations and is therefore able to infer all types statically in terms of the types and values of the input variables and outcome of control flow. Thus, with less information, it is able to statically infer types more precisely for all possible situations. This is primarily due to the fact that inferring a single type for each variable is not possible in MATLAB in the general case unless all information available during run time is available statically. FALCON has much more information available at compile time, but not enough to infer a single most general type for each of variable.

### 2.4.3 MaJIC

MaJIC, also developed at UIUC, is a MATLAB just-in-time compiler built on some of the techniques from FALCON [6, 5]. By providing a just-in-time compiler for MATLAB, MaJIC is able to preserve the interactive nature of MATLAB while achieving better performance. MaJIC utilizes two phases of compilation - a pre-phase compiler and a just-in-time compiler. The pre-phase compilation performs FALCON-style type inference. In addition, it performs additional backwards propagation of “type hints”. These hints provide likely types of the variables given their use in a few specific MATLAB operators. In this way, the analysis is able to anticipate likely arguments. Backward and forward passes alternate until the types converge. The result is a mapping from types to variables, which the compiler infers to be a likely type assignment. The code is compiled down to object code assuming these types and then stored in a code repository. The pre-compilation phase in the telescoping-languages strategy is more extensive in that, instead of trying to determine a single most-likely version, it generates code so that all possible uses of the function are covered.

During just-in-time compilation, if there is a version in the code repository that

corresponds to the given types, a call to that version is made. A version of the code is appropriate if the types of the inputs are subtypes of the actual parameter types. The variant with the most specific type signature is chosen. Ties are broken using a simple heuristic. The code repository and code selection are very similar between MaJIC and telescoping languages. However, as we will discuss in Chapter 5, the telescoping-languages compiler will use a pre-set score to determine the most beneficial variant in the event of ties.

If there is no appropriate version of the code, the just-in-time compiler performs forward type inference and compiles the code using a light-weight compiler. The resulting object code is added to the repository and may be recompiled by the more intensive compiler.

Note that the script-compiler phase could be implemented with a just-in-time compilation option, since the job of the script compiler is merely to replace the calls to the library procedures and operators with calls to the appropriate variants. Since most of the work of compilation is performed in the pre-compilation phase, the script compiler need not be a heavy-duty compiler.

#### 2.4.4 MAGICA and MATCH

MAGICA is a MATLAB type-inference system developed at Northwestern [50] and is implemented as a Mathematica application. MAGICA performs intrinsic-type inference, shape inference (sizes and ranks), and value-range inference. Type rules specify the type algebras of the primitive operations. Types are forward propagated across statements and into user-defined procedures at call sites. Type inference in this system determines a single shape that must hold for every run of the code, and therefore may be unable to determine an exact shape [51, 49]. This is insufficient to cover all possibilities for library generation in a telescoping-languages system. MAGICA solves algebraic systems of equations to determine the sizes of matrices. An advantage of this system over FALCON is that even if sizes cannot be inferred, useful information

regarding the relationships between matrix sizes is inferred. *MAGICA* uses a lattice-based approach similar to that of Kaplan and Ullman to determine a range of possible intrinsic types for each variable [48, 52]. The exact relationship of types in the range to types of all variables is still not established, which could lead to extra variant generation for a telescoping-languages system. We are unaware of any complexity results for this system.

*MATCH* (MATlab Compiler for distributed Heterogeneous computing systems) was also developed at Northwestern [8]. It requires typing directives to infer types. To perform type inference, *MATCH* converts *MATLAB* expressions into a sequence of expressions, each containing one operator. It then uses the type algebras of the *MAGICA* system to infer types.

#### 2.4.5 Menhir

Menhir is a *MATLAB* compiler that generates parallel and sequential Fortran or C code [20]. Menhir's primary feature is that it is retargetable. Menhir uses type directives at specific points in the *MATLAB* code to aid in type inference. It also clones functions and wraps conditional statements around the call sites to the function that are predicated on the variable types to avoid operator overloading. Menhir allows for extensions to the array properties analyzed, unlike other *MATLAB* compilers. Our type-inference algorithm also allows the set of array properties inferred to be easily extended. Menhir chooses the most appropriate data structure and then the most appropriate implementations to which to dispatch given the types. Choosing the implementations involves finding the implementation with the lowest casting cost. *TeleGen* can also incorporate the cost of coercion into its variant selection choice. Like telescoping languages, Menhir uses optimized libraries to achieve high-performance from the generated code. Automatically generating these libraries, however, is not handled.

#### 2.4.6 Type Inference via Partial Evaluation

Recently, Elphick et al. implemented a type inference system for MATLAB that relies on partial evaluation [35]. Because the calling contexts for library procedures are unknown at library compilation time, this strategy provides little benefit for telescoping languages.

#### 2.4.7 Cost of Interpretation

Menon and Pingali performed a study of the cost of the overhead of the MATLAB interpreter along with a study of some source-level transformations [63]. It was found that type optimization, elimination of array-bounds checking, and elimination of dynamic reallocation of arrays all had dramatic effects on the performance of the application. The type-inference algorithm presented in this dissertation provides enough type information to be able to perform these optimizations.

#### 2.4.8 Other Parallel MATLAB Compilers

The Otter system, developed at Oregon State University, is designed to compile MATLAB into C with MPI [70]. They perform intrinsic type and rank inference as much as possible at compile time. Type information is extracted in part from input files. Size inference is left for runtime evaluation.

Compiling MATLAB for parallel architectures is beyond the scope of this dissertation, although we do describe how to extend type inference to infer data distribution. However, any MATLAB compiler must have a way of dealing with the dynamic-typing problem. Most parallel MATLAB compilers infer types using a system similar to the one in FALCON and are therefore not mentioned here.

#### 2.4.9 MATLAB-Like Languages

Octave is very similar to MATLAB [34]. Because it is open-source, and therefore has accessible libraries to implement the primitive operations, octave libraries may



be used to replace the MATLAB primitive operators in both library compilation and script compilation.

## Chapter 3

# Type Inference

Type information drives specialization in TeleGen, where *type* refers to properties of the variables, such as intrinsic type and matrix size. We chose to start with type-based specialization in the prototype compiler because knowledge of types is essential for performing many of the optimizations found in high performance compilers. Because we chose MATLAB as our source language, type inference becomes necessary to correctly translate to lower-level languages. However, we envision that technologies developed in this dissertation will be beneficial to all future applications of telescoping languages regardless of the source language. This is especially true since the type-inference algorithm described in this chapter can be applied to a broad range of analysis problems.

### 3.1 Motivating Type Inference

Type inference must be performed to translate from a high-level, dynamically-typed language such as MATLAB to a lower-level, explicitly-typed language such as Fortran or C. Furthermore, several specialization opportunities may appear from understanding the possible types of the variables.

In the telescoping-languages strategy, analysis of library procedures occurs before the calling context is available. Pre-compiling libraries, when the calling contexts are unknown, has traditionally led to limited specialization opportunities. The telescoping-languages strategy aims to address this problem by generating several variants for a given script, with each variant intended for a different possible configuration of types over the arguments. Therefore, the telescoping-languages strategy

achieves the benefit of specializing for actual input types without having to wait until the calling routines are available. Note that, while important to the success of the telescoping-languages strategy, type inference and specialization alone are not sufficient to achieve the highest performance possible. For example, constant propagation is an important optimization that cannot be represented as a type inference problem. The telescoping-languages strategy gives up the opportunity to perform constant propagation across call sites.

To provide specialization opportunities for all calling contexts, the traditional methods of type inference, whereby a single type or a set of possible types is assigned to every variable or expression, will not suffice. In the case where each expression is assigned a single type, there is not enough specialization opportunity. In the case where each expression is assigned multiple types, there is not enough precision, resulting in too many variants.

Because the prototype compiler uses MATLAB as the source language, type inference becomes even more complicated. For example, it is usually impossible to infer a single type for every variable that will suffice for every possible dynamic invocation of a MATLAB procedure. Note that the features of MATLAB that complicate type inference can be found in several other high-level, domain-specific languages, such as R, Python, or LabVIEW.

To address these problems, we developed a new type-inference strategy that is able to infer all valid type assignments or type configurations for the procedure without knowledge of the possible calling contexts. The type-inference algorithm serves two purposes. First, it determines the minimum number of type configurations that could legally occur in any dynamic invocation of the procedure. Second, type inference is used to statically determine, for each corresponding variant, which optimized implementations should be invoked at each call site.

This chapter defines and formalizes the solution to type inference in telescoping languages. Type inference over a procedure produces a *type jump-function*, which

lists every possible type configuration over the variables in terms of the parameter types.

Type inference over a procedure produces a *return type-jump-function*, or *procedure type*, which lists every possible type configuration over the variables in terms of the input types. To represent this information, we introduce a new notion of types that we term *mutually exclusive types*, which are related to intersection types in that multiple type configurations over the parameters are represented in the principal type of a procedure. These types enable us to express properties that are necessary to accurately infer types in a polymorphic, array-based language such as MATLAB.

To formally describe these types, we present a core calculus for MATLAB and impose a static type system, which we prove is type safe. We then formalize the constraint system and define the conditions under which these constraints are satisfied. We prove that these conditions are sound and complete with respect to the type system.

Although inference of intersection types is undecidable in general, we prove that the inference algorithm for mutually-exclusive types is solvable in polynomial time under a set of simplifying conditions. We argue that these conditions typically hold in practice.

## 3.2 The Type-Inference Problem

The type-inference algorithm presented in this dissertation must serve two purposes. First, it must determine the minimum number of valid type configurations or type assignments. Second, it must provide the information necessary to statically determine, for each variant, which optimized implementations should be dispatched at each call site.

Because we use MATLAB as our base language, the type-inference problem is complicated by the fact that MATLAB has a high degree of polymorphism. For example, the `*` operation in MATLAB is overloaded based on the types of the inputs. This

operation can perform scalar multiplication, matrix scaling, or matrix multiplication depending on the rank of the inputs. It is therefore necessary to precisely infer the types to preserve the semantics of the code when we translate MATLAB operations to calls to libraries in Fortran or C.

Furthermore, the `*` operation is used for both real and complex inputs. The performance benefit from correctly determining the types and calling the implementation that is optimized for those types can be large [19].

Type inference must be precise enough to preserve the semantics of the code and provide specialization opportunities.

To describe our type-inference mechanism in the context of a concrete language, we consider a subset of MATLAB and ignore some of its rarely-used features such as dynamic evaluation of strings as code, object-oriented features, and structures. In Chapter 4, we describe extensions to the basic type-inference strategy that enable type inference to be applied to a broader subset of MATLAB as well as other languages.

### 3.2.1 Type Problems

To carry out specialization based on variable types, we first define the variable properties that are of interest. In general, the properties should be such that they can be encoded in the target language and the compiler for that language can leverage the information for optimization. They should also reflect the power of the source language.

Since MATLAB is an array-based language, we use the 4-tuple definition of a variable type based on work by deRose [33]. We define a type to be a tuple  $T = \langle \tau, \delta, \sigma, \pi \rangle$  where,

$\tau$  is the *intrinsic type* of the variable (e.g., integer, real, complex). The intrinsic types constitute a fixed set of scalar types. Solving  $\tau$  solves the “intrinsic type problem.”

$\delta$  is the upper bound on the *number of dimensions* for an array variable, also

called the rank. A tighter bound can be reached when the type inference system determines that the variable has a size of 1 in one or more dimensions.  $\delta$  will always be greater than or equal to 2.

$\sigma$  is a tuple showing the maximum *size* of an array variable in each possible dimension.  $\sigma^A = \langle 1, 1 \rangle$  means that A is a scalar. Since some of the sizes may be 1,  $\sigma$  also determines the actual rank of the variables (as opposed to a maximum number of dimensions). Solving  $\sigma$  solves the “size problem.”

$\pi$  is the *sparsity pattern* of an array variable (*e.g.*, dense, triangular, symmetric, etc.). Solving  $\pi$  solves the “pattern problem.” We assume that the lattice for the pattern inference problem is finite.

This list can be extended as needed for other languages and other problems. In this chapter, we focus on the size-inference problem to demonstrate the power of our type-inference algorithm

### 3.2.2 Type Inference in MATLAB

MATLAB’s simplicity makes it popular among scientific programmers. However, some of the very features that make MATLAB a desirable language for programming make it difficult for the compiler to translate to lower-level languages. Some of these features include:

1. MATLAB is dynamically-typed. This makes inferring primitive types necessary to generate code in lower-level languages such as Fortran or C, both of which require explicit typing.
2. Operators are heavily overloaded. For example, the  $*$  operation can refer to both matrix-matrix multiplication and matrix scaling, depending on whether an operand is scalar. Therefore, determining whether a variable is a scalar or not is important for correctness.

3. Variables can change types in the middle of the program, including arrays growing in the middle of a loop. Inferring the maximum number of iterations of a loop would help avoid reallocating the array at every iteration.
4. All variables are treated as arrays, including scalars, which are  $1 \times 1$  arrays. Overloaded operators such as  $*$  are overloaded based on the size or extent of the array in each dimension.

These features not only make type inference essential for compilation, they also make it difficult to statically determine the types of the variables.

### 3.2.3 Type Inference in Telescoping Languages

Since, in the telescoping-languages framework, libraries are preprocessed before calling context is known, type inference must use information within the library procedures to determine a minimum number of type configurations.

Information about the type of a variable in a MATLAB procedure depends on the following:

1. The operation that *defines* the variable. For example, in the statement

$A=w*v,$

$A$ 's size can be determined from the sizes of  $v$  and  $w$ .

2. The operations that use the variable, since operations impose type restrictions on their inputs. For example, in the statement:

$H(1,1) = \text{alpha},$

$\text{alpha}$  must be scalar since it is assigned to a single element of  $H$ .

The first causes *forward propagation* of variable properties along the control flow, while the second causes *backward propagation*. The type-inference system must infer types in both directions for the most precise outcome.

### 3.2.4 Existing Solutions

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

There are two main difficulties to using a traditional data-flow-analysis framework for the problem described here.

1. It is difficult, if not impossible, to determine that the analysis will halt for a given subroutine. This is primarily due to the fact that for size inference, the finite-descending-chain property does not hold.
2. The compiler must find all type solutions allowed by a procedure. Therefore, data-flow analysis is ill-suited for the problem, since if the analysis converges, it converges to a single solution (as in FALCON) or to a set of types as in the algorithm presented by Aho, Sethi and Ullman [4]. In the first case, there is not enough specialization opportunity, since the types are too general. In fact, in many cases a type will not be assigned, since the possible types might not have a most general type that can represent all possibilities. When this occurs in Falcon, types must be inferred in the dynamic compilation phase.

In the second case, the exact relationship of the types of variables is not captured, resulting in more type configurations than could validly occur in the program. For example, the data-flow analysis may be able to infer that an input variable and an output variable to an operation that performs transpose can each be either a upper or lower triangular, but it would not be able to capture the fact that the output is only an upper triangular if the input is a lower triangular. Such imprecision could lead to a radical increase in the number of variants, and with it, a corresponding increase in library compiler times.



If the telescoping-languages library compiler were generating variants from the data-flow solution, it would generate an extra variant that would provide for the possibility that an upper triangular input might produce an upper triangular output, which is not a possibility. To avoid generating superfluous variants, the compiler needs a more precise type-inference strategy.

### 3.3 Type-Inference Solution for Size Problem

To perform type analysis over MATLAB procedures in the telescoping languages framework, we developed an alternative solution to the traditional data-flow techniques that performs analysis over the procedure as a whole [62]. This section gives an overview of the solution to provide the necessary background for the rest of the dissertation.

To illustrate the type-inference algorithm, we describe the solution to the size-inference problem, since it has the most interesting properties. Size inference solves two problems, the rank problem (scalar or non-scalar) and the size or extent of each dimension of the matrix. The rank problem is necessary to understand which operators and procedure calls are intended, since operators are overloaded based on the rank. The extent in each dimension is necessary to understand the relationship of variable sizes in the library procedure, which is important for optimizations such as preallocating arrays based on the sizes of the inputs. To isolate the size-inference problem, we assume that the number of dimensions is always two (scalars have a size of one in each dimension). We discuss how to handle multiple dimensions as well as how to use the type-inference algorithm to infer intrinsic types and sparsity pattern in Section 4.

We perform type inference over an intermediate representation of the MATLAB code in which all expressions have been expanded so that the results of each operation or procedure call are assigned to variables. Because each statement now contains only one operator or procedure call, we use these terms interchangeably.

Also, to handle the fact that a variable may change type in the middle of a

procedure, we assume that the procedures have been converted to SSA form [28] so that each use of a variable refers to a single definition. Redefinition of a section of an array results in a new array, since this can cause a change in type. The arrays and variables will be re-merged during code generation if possible.

Finally, we assume for simplicity that all global variables have been converted to inputs and outputs to the procedures. Therefore, when we refer to the number of parameters, we are actually referring to the number of parameters plus the number of global variables.

The type-inference algorithm we propose determines the information that each individual operation or procedure call gives about the types of the variables involved and then combines that information over the entire procedure to find types. Thus, forward and backward inference occur simultaneously.

### 3.3.1 Statement Information

The information from operations is given in the form of propositional constraints on the types of variables involved in the statement, called *statement constraints*. MATLAB operations, and typical library procedures, are heavily overloaded based on the types of the inputs. Therefore, statement constraints need to represent all possible valid type configurations.

Statement constraints are formed using a database of *return type-jump-functions* containing one entry per procedure or operation. Figure 3.1 shows an example of the return type-jump-function for the size problem on the MATLAB multiplication operation, “\*”. The *clauses* representing possible ranks over the input and output parameters are composed through logical disjunction. These clauses are defined to be mutually exclusive, thereby imposing the property that each clause represents a distinct type configuration. The clauses are made up of size definitions for each parameter. The sizes are represented by \$-variables, which give the exact size relationships between the variables. The parenthetical expression in each clause details

```

o1 = i1 * i2

1  $\sigma^{o1} = \langle 1, 1 \rangle \wedge \sigma^{i1} = \langle 1, 1 \rangle \wedge \sigma^{i2} = \langle 1, 1 \rangle$  XOR
2  $\sigma^{o1} = \langle \$1, \$2 \rangle \wedge \sigma^{i1} = \langle 1, 1 \rangle \wedge \sigma^{i2} = \langle \$1, \$2 \rangle \wedge (\$1 \neq 1 \vee \$2 \neq 1)$  XOR
3  $\sigma^{o1} = \langle \$1, \$2 \rangle \wedge \sigma^{i1} = \langle \$1, \$2 \rangle \wedge \sigma^{i2} = \langle 1, 1 \rangle \wedge (\$1 \neq 1 \vee \$2 \neq 1)$  XOR
4  $\sigma^{o1} = \langle \$1, \$3 \rangle \wedge \sigma^{i1} = \langle \$1, \$2 \rangle \wedge \sigma^{i2} = \langle \$2, \$3 \rangle \wedge ((\$1 \neq 1 \wedge \$2 \neq 1) \vee$ 
    $(\$3 \neq 1 \wedge (\$1 \neq 1 \vee \$2 \neq 1)))$  XOR
5  $\sigma^{o1} = \langle 1, 1 \rangle \wedge \sigma^{i1} = \langle 1, \$1 \rangle \wedge \sigma^{i2} = \langle \$1, 1 \rangle \wedge (\$1 \neq 1)$ 

```

Figure 3.1 : Example of a procedure type for the size inference problem on the MATLAB “\*” operation. Each clause gives a possible size configuration over the sizes of the variable. The first clause states that the operation could be scalar multiplication. This second and third clauses state that “\*” could be a scaling operation. The fourth clause shows the operation could be matrix-matrix or matrix-vector multiplication. The last clause gives the possibility that the operation is multiplication of two vectors. This last case is necessary to keep track of the fact that *o1* may be scalar when *i1* and *i2* are not. While the last clause may not represent different functionality from the fourth clause for this “\*” operation, the output may be used in another statement, where understanding that it is scalar may be important to determining the meaning of the operation.

the conditions necessary to ensure that each clause is mutually exclusive. Note that the clauses aren’t only tracking rank, but whether the variables are row or column vectors. The constraint language is formalized in Section 3.5.

The *\$-variables* are simply place holders for integer values representing sizes. Since each *\$-variable* can be used for multiple variable sizes within a single statement constraint, they capture the size relationships between the variables in a single operation. The fields in  $\sigma$  can be defined to be linear expressions over the *\$-variables*. Thus far, these expressions are sufficient to represent the size relationships between the variables. For example, to represent concatenation of two array dimensions with sizes *\$1* and *\$2*, the concatenated dimension would have size  $\$1 + \$2$ . *\$-variables* are used to track the extents of the matrices in each dimension.

Notice that in addition to tracking array sizes through the use of *\$-variable* place holders, the clauses also track whether the source-code variables are scalars or non-scalars (*i.e.*, the rank problem). This is important because MATLAB operators are

overloaded on this information. For example, the `*` operator has completely different semantics depending on whether its arguments are scalars or not. Further differentiation between number of dimensions does not demonstrate these drastic effects on the meaning of the operators and therefore does not need to be represented in separate clauses. We describe how to use the type-inference algorithm to infer dimensionality in Chapter 4.

The statement constraint formed from this procedure type at a particular application site is identical to the procedure type except that the variables are replaced by the actual input and output parameters. Since the statement constraints should be formed in isolation from each other,  $\$$ -variables in a statement constraint should be interpreted as though they are existentially quantified. Rather than including binding constructs, we ensure that  $\$$ -variables are not shared across statement constraints.

$\$$ -variables may be replaced by constants in a statement constraint, when it can be determined that the matrix size is constant in a particular dimension for the statement in the MATLAB program. This can cause a reduction in the number of clauses in the statement constraint, since the constant may guarantee that certain clauses can never hold. The size of a matrix may also depend on a variable in the procedure. Unlike the case of constants, since the compiler may not be able to determine that a program variable is never 1, the clauses stating the possibility that the size may be 1 must be maintained.

### 3.3.2 Combining Statement Information

In a valid procedure, the type of a variable must satisfy all the constraints imposed by all the operations that can feasibly execute in any run of the program. Therefore, the set of valid type configurations over the local variables in the procedure, which we call the *type jump-function* or, more formally, the *principal type*, can be determined by taking the conjunction of the statement constraints over the procedure, called the *procedure constraint*, and finding all possible type configurations of the variables that

	config A	config B	config C
$\sigma^{A_1}$	$\langle 1, 1 \rangle$	$\langle 1, 1 \rangle$	$\langle \$1, \$1 \rangle$
$\sigma^{v_1}$	$\langle 1, 1 \rangle$	$\langle \$1, 1 \rangle$	$\langle \$1, 1 \rangle$
$\sigma^{k_1}$	$\langle 1, 1 \rangle$	$\langle 1, 1 \rangle$	$\langle 1, 1 \rangle$
$\sigma^{v_2}$	$\langle 1, 1 \rangle$	$\langle \$1, 1 \rangle$	$\langle \$1, 1 \rangle$
$\sigma^{w_1}$	$\langle 1, 1 \rangle$	$\langle \$1, 1 \rangle$	$\langle \$1, 1 \rangle$
$\sigma^{\alpha_1}$	$\langle 1, 1 \rangle$	$\langle 1, 1 \rangle$	$\langle 1, 1 \rangle$
$\sigma^{f_1}$	$\langle 1, 1 \rangle$	$\langle \$1, 1 \rangle$	$\langle \$1, 1 \rangle$
$\sigma^{c_1}$	$\langle 1, 1 \rangle$	$\langle 1, 1 \rangle$	$\langle 1, 1 \rangle$
$\sigma^{f_2}$	$\langle 1, 1 \rangle$	$\langle \$1, 1 \rangle$	$\langle \$1, 1 \rangle$
$\sigma^{\alpha_2}$	$\langle 1, 1 \rangle$	$\langle 1, 1 \rangle$	$\langle 1, 1 \rangle$
$\sigma^{V_1}$	$\langle 1 \rangle$	$\langle \$1 \rangle$	$\langle \$1 \rangle$
$\sigma^{f_3}$	$\langle \$1, 1 \rangle$	$\langle \$1, 1 \rangle$	$\langle \$1, 1 \rangle$
$\sigma^{\beta_1}$	$\langle 1, 1 \rangle$	$\langle 1, 1 \rangle$	$\langle 1, 1 \rangle$
$\sigma^{v_3}$	$\langle \$1, 1 \rangle$	$\langle \$1, 1 \rangle$	$\langle \$1, 1 \rangle$
$\sigma^{V_2}$	$\langle \$1 \rangle$	$\langle \$1 \rangle$	$\langle \$1 \rangle$
$\sigma^{w_2}$	$\langle \$1, 1 \rangle$	$\langle \$1, 1 \rangle$	$\langle \$1, 1 \rangle$
$\sigma^{h_1}$	$\langle k_1, 1 \rangle$	$\langle k_1, 1 \rangle$	$\langle k_1, 1 \rangle$
$\sigma^{f_4}$	$\langle \$1, 1 \rangle$	$\langle \$1, 1 \rangle$	$\langle \$1, 1 \rangle$
$\sigma^{c_2}$	$\langle k_1, 1 \rangle$	$\langle k_1, 1 \rangle$	$\langle k_1, 1 \rangle$
$\sigma^{f_5}$	$\langle \$1, 1 \rangle$	$\langle \$1, 1 \rangle$	$\langle \$1, 1 \rangle$
$\sigma^{f_6}$	$\langle \$1, 1 \rangle$	$\langle \$1, 1 \rangle$	$\langle \$1, 1 \rangle$
$\sigma^{V_3}$	$\langle \$1 \rangle$	$\langle \$1 \rangle$	$\langle \$1 \rangle$
$\sigma^{h_2}$	$\langle k_1, 1 \rangle$	$\langle k_1, \$1 \rangle$	$\langle k_1, 1 \rangle$

```

function [V, H, f] =
  ArnoldiC(A1, k1, v1);
v2 = v1/norm(v1);
w1 = A1*v2;
alpha1 = v2*w1;
tmp1 = v2*alpha1;
f1 = w1 - tmp1;
c1 = v2'*f1;
tmp2 = v2*c1;
f2 = f1 - tmp2;
alpha2 = alpha1 + c1;
V1(:, 1) = v2;
H1(1, 1) = alpha2;
for j = 2 : k1,
  f3 = phi(f2, f5);
  beta1 = norm(f3);
  v3 = f3/beta1;
  H2(j, j-1) = beta1;
  V2(:, j) = v3;
  w2 = A1*v3;
  h1 = V2(:, 1:j)'*w2;
  tmp3 = V2(:, 1:j)*h1;
  f4 = w2 - tmp3;
  c2 = V2(:, 1:j)'*f4;
  tmp4 = V2(:, 1:j)*c2;
  f5 = f4 - tmp4;
  h2 = h1 + c2;
  H3(1:j, j) = h2;
end
V3 = phi(V1, V2);
H4 = phi(H1, H3);
f6 = phi(f2, f5);

```

Figure 3.2 : The resulting type configurations and the corresponding pruned SSA form of ArnoldiC.

satisfy the resulting boolean expression. Solving typical propositional constraints of this form is NP-hard.<sup>1</sup> However, we show in Section 3.4 that under certain conditions

<sup>1</sup>The well known 3-SAT problem can be reduced to this problem [27].

that occur most frequently in practice, we can devise an efficient algorithm using the specific properties of the problem.

The type jump-function inferred by our type-inference algorithm from the MATLAB procedure, `ArnoldiC`,<sup>2</sup> is shown in Figure 3.2 in tabular form. Each column represents a different possible type configuration for the SSA form of `ArnoldiC`.

In this example, the types of variables can be exactly inferred from the type configurations table given the inputs. For example, if  $A_1$  is non-scalar, then configuration C should be used. Configuration C is, in fact, the only type configuration intended by the library writers. An annotation stating that the input  $A$  is never scalar would have made configuration C the only result of type inference. The telescoping-languages strategy provides a mechanism for such annotations as will be discussed in Section 3.4.

Because the only free variables occurring in each configuration are the runtime values  $\$1$  (the size of  $A_1$  in each dimension) and the iteration variable  $j$  (whose maximum size is  $k_1$ ), the sizes of all the variables in the procedure can be determined once the sizes and values of the inputs are known. The type-inference algorithm is powerful enough to infer that  $A_1$  must be square (*i.e.*, that it has the same size in both dimensions) if it is a matrix, as well as the fact that many of the variables are vectors.

Note that the type configurations in the table are mutually exclusive and jointly exhaustive, and they involve the fewest number of  $\$$ -variables possible, so that the fewest values need to be known for all the sizes to be known.

### 3.4 An Implementation for Solving Procedure Constraints

In this section, we give an efficient implementation for solving procedure constraints and prove that it is efficient in practice. To simplify the initial description, we first describe the type-inference algorithm for straight-line code. In Chapter 4, we extend

---

<sup>2</sup>This procedure is from the ARPACK development code.

the algorithm to handle the other constructs, such as control flow, which are necessary for applications that occur in practice.

Solving constraints can be broken down into three phases - building a graphical representation of the constraints, finding  $n$ -cliques over the graph, and solving the cliques to produce the procedure type.

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

1. The type-inference engine has valid code on input (*i.e.*, all variables are defined before being used).<sup>3</sup>
2. All global variables have been converted to input and output parameters.
3. The number of input and output parameters in each operation or procedure is bounded by a constant. This property is important to limit the complexity and is common in practice since parameter lists do not tend to grow with the size of the procedure [23]. Also, libraries are not typically written with global variables to ensure that the procedures function in arbitrary contexts.

### 3.4.1 Reducing Type Inference to Clique Finding

After constraints for each operation have been determined, the compiler must reason about them over the whole procedure. That is, it must find all possible type configurations that satisfy the whole-procedure constraint. By representing the operation constraints as nodes in a leveled-graph, where each node is associated with a level and each level corresponds to a different operation, the problem is reduced to finding  $n$ -cliques, where  $n$  is the number of operations in the procedure, and an  $n$ -clique is a complete subgraph involving  $n$  nodes.

---

<sup>3</sup>This is a reasonable assumption for MATLAB programs since users can develop and test their code in the MATLAB interpreter before giving it to the optimizing compiler. It is also easy to verify since SSA construction fails if there is an undefined use.

$$\begin{aligned}
& \mathbf{A = b + c} \\
\mathbf{1a} & \sigma^A = \langle 1, 1 \rangle \wedge \sigma^b = \langle 1, 1 \rangle \wedge \sigma^c = \langle 1, 1 \rangle \quad \text{XOR} \\
\mathbf{1b} & \sigma^A = \langle \$1, \$2 \rangle \wedge \sigma^b = \langle 1, 1 \rangle \wedge \sigma^c = \langle \$1, \$2 \rangle \wedge (\$1 \neq 1 \vee \$2 \neq 1) \quad \text{XOR} \\
\mathbf{1c} & \sigma^A = \langle \$1, \$2 \rangle \wedge \sigma^b = \langle \$1, \$2 \rangle \wedge \sigma^c = \langle 1, 1 \rangle \wedge (\$1 \neq 1 \vee \$2 \neq 1) \quad \text{XOR} \\
\mathbf{1d} & \sigma^A = \langle \$1, \$2 \rangle \wedge \sigma^b = \langle \$1, \$2 \rangle \wedge \sigma^c = \langle \$1, \$2 \rangle \wedge (\$1 \neq 1 \vee \$2 \neq 1) \\
\\
& \mathbf{E = c - d} \\
\mathbf{2a} & \sigma^E = \langle 1, 1 \rangle \wedge \sigma^c = \langle 1, 1 \rangle \wedge \sigma^d = \langle 1, 1 \rangle \quad \text{XOR} \\
\mathbf{2b} & \sigma^E = \langle \$3, \$4 \rangle \wedge \sigma^c = \langle 1, 1 \rangle \wedge \sigma^d = \langle \$3, \$4 \rangle \wedge (\$3 \neq 1 \vee \$4 \neq 1) \quad \text{XOR} \\
\mathbf{2c} & \sigma^E = \langle \$3, \$4 \rangle \wedge \sigma^c = \langle \$3, \$4 \rangle \wedge \sigma^d = \langle 1, 1 \rangle \wedge (\$3 \neq 1 \vee \$4 \neq 1) \quad \text{XOR} \\
\mathbf{2d} & \sigma^E = \langle \$3, \$4 \rangle \wedge \sigma^c = \langle \$3, \$4 \rangle \wedge \sigma^d = \langle \$3, \$4 \rangle \wedge (\$3 \neq 1 \vee \$4 \neq 1)
\end{aligned}$$

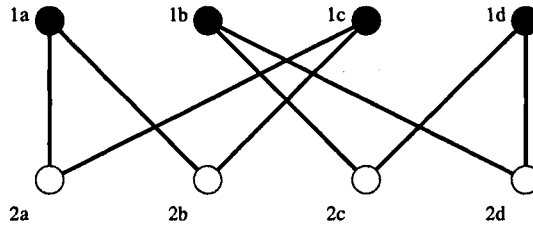


Figure 3.3 : Example graph.

Figure 3.3 shows a simple example of how the graph is constructed. Each clause, or possible type configuration for an operation, is represented by a node at the level corresponding to the lexicographical position of the statement in the code that contains it. There is an edge from one node to another if the expressions in the nodes do not contradict each other. For example, there are no edges from  $1b$  to  $2b$  since  $c$  cannot be both scalar and non-scalar. Note that since each clause is mutually exclusive, there is no edge between nodes on the same level.

The final graph has  $n$  levels, where  $n$  is the number of operations or procedure calls (or statements in the expanded form). Each level is bounded by  $2^v$  nodes, since 2 is the number of possible types for each variable (scalar or non-scalar), and  $v$  is the number of variables involved in the statement.  $v$  is assumed to be small by the third assumption.  $2^v$  is the number of possible type configurations, or entries in the type table, for the variables in the operation corresponding to that level, since there



are two possibilities for each variable. Since  $v$  is assumed to be bounded by a small constant,  $2^v$  must be bounded by a constant.

The goal of our type-inference algorithm is to find the set of type configurations or type assignments of the variables that could validly (without type errors) occur in practice. Because the clauses in the statement constraint represent every possible type assignment of arguments at the call site, the set of type configurations over the procedure must satisfy one clause from every statement constraint or one node on each level of the graph. Therefore, finding the set of all possible type configurations maps naturally to the set of all  $n$ -cliques (where  $n$  is the number of levels in the graph, and a clique is a complete subgraph) over the graph.

### 3.4.2 Finding Cliques

To show that using cliques to determine types is viable for our problem, we must prove that we can find cliques efficiently. First, we prove that the number of type configurations is bounded by a small number, and then from this, prove that the total number of  $n$ -cliques is bounded by a small number.

Let  $u$ -vars be defined to be the smallest set of variables such that all other variable types can be determined from the types of the  $u$ -vars.  $U$ -vars represent the set of variables that are not guaranteed to be statically determinable in terms of the types of the other variables. Examples of  $u$ -vars are input types, merged types at control-flow join points, operations for which there is no available return type-jump-function, and the results of non-type-input-dependent operations (where *type-input-dependent* means that the types of the outputs can be uniquely determined from the types of the inputs). The algorithm may be able to infer exact types for some or all of the  $u$ -vars by their uses. For the simple case where all operations are type-input-dependent and there is no complex control flow,  $u$  is bounded above by the number of input parameters and is therefore small by assumption 3.

We define a *valid* procedure to be one with the property that all variable definitions

occur lexically before any of their uses.

**Theorem 3.4.1** *The number of possible type configurations is bounded by  $l^u$ , where  $u$  is an upper bound on the number of u-vars and  $l$  is the number of possible types for each variable. For the size inference problem,  $l$  is 2.*

Proof: By the definition of u-vars, all other variable types excepting the u-vars can be statically determined in terms of the types of the u-vars. Therefore, the total number of possible configurations over all the variables is just the number of possible configurations of the u-var types. For the size problem, this is  $2^u$  since each u-var could potentially be either scalar or non-scalar and the clauses are mutually exclusive. The  $\$$ -variables do not change the number of type configurations, since they do not affect the number of clauses in the statement constraint, but will be used to solve for actual sizes once the ranks have been inferred.  $\square$

**Theorem 3.4.2** *The number of n-cliques is bounded by  $l^u$ .*

Proof (by contradiction): Since each clique represents a possible type configuration, we need to show that no two cliques represent the same type configuration. We start by assuming there are two distinct cliques that represent the same type configuration over the variables. The cliques must differ at one or more levels to be distinct. Since expressions in nodes of the same level contradict each other, at least one variable must have a different type. Therefore, the two cliques cannot have the same type configuration over the variables.  $\square$

Finding n-cliques in general is  $\mathcal{NP}$ -Complete [27]. However, we claim that given the structure of the problem, we are solving a subset of the n-clique problem that has a polynomial time solution given a bound on  $u$ .

The solution must be able to take advantage of the specific properties of the problem. Figure 3.4 gives a simple iterative algorithm to achieve this. The algorithm starts with one level and puts each node in that level in its own clique. For each

```

findCliques
input:  graph G
output: CurrCliques
initialize CurrCliques to first-level nodes
1 for every level r in G - first row
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       if (candidate)
9         newCliques = newCliques + clique(c,n)
10  CurrCliques = newCliques

```

Figure 3.4 : Iterative n-clique finding algorithm.

subsequent step, it compares each node in the current level with each already formed clique. If the node has an edge to every member of the clique, it forms a new clique with the old clique. It does this until it reaches the last level.

The loop starting on line 4 in Figure 3.4 iterates over the cliques from the previous step. Because the bound of  $2^u$  only holds for the final number of cliques, we need to find a bound on the number of intermediate cliques to limit the complexity.

**Theorem 3.4.3** *The number of cliques at each step of the iterative n-clique-finding algorithm is bounded by  $l^u$  if the levels are visited in program order.*

Proof: The number of cliques for a program with only arguments and an empty body is the number of type configurations over the input and output arguments, or  $l^u$ . We have from above that on valid procedures there is an upper bound of  $l^u$  on the number of cliques. At any operation or procedure call, the rest of the code can be left off and the remaining (beginning) code is still valid.<sup>4</sup> Therefore, since every

---

<sup>4</sup>Since valid only refers to the fact that every variable is defined before being used, SSA gives us this property in the presence of control-flow constructs.

iteration of the algorithm has processed valid code, if the levels are in program order, after every iteration, the algorithm will have produced cliques on valid code. The number of cliques after every iteration must be bounded by  $l^u$ .  $\square$

With  $l^u$  cliques after every iteration, the n-clique-finding algorithm takes  $l^v l^u n^2$  steps, where  $n$  is the number of operations,  $l^v$  is the maximum number of nodes in a level (always bound by a small constant), and  $u$  is the number undeterminable variables. Therefore, the overall time complexity is  $O(n^2)$  if  $l^u$  is bounded by a constant, which is true when all operations are type-input-dependent (*i.e.*, all output types can be determined by the types of the inputs) and there is no complex control flow.

Of course, in MATLAB, not all operations are type-input-dependent. For some operations, the types of the outputs could depend on the values of the inputs rather than the types. This means that an operation could produce multiple output types on a given set of input types. Variables defined by such operations are u-vars.  $u$  should still remain small, since few operations are not type-input-dependent. Note that the algorithm works even without this assumption, but the complexity could become exponential in the worst case.

### 3.4.3 Solving Cliques

Each clique represents a different possible type configuration. However, the equations in each clique still need to be solved to determine a more succinct representation of the type configurations. Since each node in a clique has equations that use different  $\$$ -variables from equations in other nodes, the exact size relationships are still not apparent. To reduce the cliques to a more useful form, the compiler must “solve” the equations so that the sizes of the variable are all expressed in terms of the sizes of the u-vars and the minimum number of  $\$$ -variables are used. The last requirement guarantees that the exact relationship between the variable sizes is explicit.

The following strategy provides a general solution to the clique-solving problem. We implemented a strategy that solves a specific instance of the problem in which there are no coefficients. The strategy for finding a general solution is a relatively straightforward extension of the implemented strategy.

Because the size in each dimension is written in terms of a linear expression over the \$-variables, solving the equations in the clique can be formulated as solving systems of linear Diophantine equations. Note that the system will most likely be under-constrained, in which case the solver must solve the equations in terms of the minimum number of variables (in this case, \$-variables).

Each variable dimension is solved separately. Therefore, equations are of the form:

$$v_d = \sum_{i=0}^n a_i x_i + b,$$

where  $b$  is a constant, the  $x_i$ 's range over the set of \$-variables,  $v$  is a variable, and  $d$  is a dimension of  $v$ .

First, the equations are sorted into systems of equations so that each system contains all equations that define the size for a single dimension of a variable. For each system, the solver performs a linear pass over the equations. If there is only a single equation, no further action is necessary. Otherwise, the solver examines two equations from the system at a time and solves for the \$-variables in the equations. The solution from this solving process is guaranteed to involve fewer \$ variables than the original equations. The solutions to the \$-variables are then substituted for the \$-variables in all the equations from all the systems. After \$-variables substitution, the two examined equations are the same, so one of the equations is removed from the system. The solver then solves this new equation with the next equation from the system if there is one available. When this process is completed, there will be a single equation defining each dimension of each variable. These equations will involve the fewest number of \$-variables needed to represent all possible size configurations.

To solve for the \$-variables in two equations, the solver subtracts the equations to get rid of the procedure variable. The resulting equation is of the form:

$$\sum_{i=0}^n a_i x_i = b.$$

**Theorem 3.4.4** All solutions to the equation  $\sum_{i=0}^n a_i x_i = b$  have the form:

$$x_0 = x'_0 - \sum_{j=1}^n \frac{a_j k_j}{\gcd(a_0, a_j)}$$

$$x_i = x'_i - \frac{a_0 k_i}{\gcd(a_0, a_i)}, \forall 1 \leq i \leq n,$$

where  $x_i = x'_i \forall i, 0 \leq i \leq n$  is one solution to the equation.

Proof: Substituting the general solution into the original equation, we get

$$a_0 \left\{ x'_0 + \sum_{j=1}^n \frac{a_j k_j}{\gcd(a_0, a_j)} \right\} + \sum_{j=1}^n a_j \left( x'_j - \frac{a_0 k_j}{\gcd(a_0, a_j)} \right) = b.$$

Simplifying, we get

$$\sum_{i=0}^n a_i x'_i + a_0 \sum_{j=1}^n \frac{a_j k_j}{\gcd(a_0, a_j)} - a_0 \sum_{j=1}^n \frac{a_j k_j}{\gcd(a_0, a_j)} = b.$$

Therefore,

$$\sum_{i=0}^n a_i x'_i = b,$$

which we know to be true based on the definitions of the  $x'_i$ 's.  $\square$

The  $k_i$ 's are fresh \$-variables. Thus, given two equations from the cliques, the solver is able to reduce the number of \$-variables from the equations by one. The original \$-variables (the  $x_i$ 's) are replaced by the solution in all the equations in which they occur, reducing the total number of \$-variables over all the equations by one. Continuing in this fashion will eventually yield the smallest number of \$-variables needed to compute all the sizes.

One important feature of this strategy is that in the final solution, each \$-variable appears in at least one equation by itself. This property held true before solving, and

each step of solving preserved this property. The equations can be arranged so that each  $\$$ -variable appears by itself in a size equation defining the type of a u-var. Thus, the property that all the variable sizes are computed in terms of the sizes of the u-vars with a small amount of back-solving to isolate the  $\$$ -variables from the coefficient and the  $x'_i$  is maintained. The u-vars themselves may be statically inferred. In some cases, it is determined that a clique is invalid if there is no solution from the solver.

Now the only requirement for a solution to the equations is a starting solution. This can be calculated from the computation of  $gcd(a_0, \dots, a_n)$ , which we will call  $gcd^n$  for brevity. It is well-known that given integers  $a, b$  there are integers  $s$  and  $t$  such that  $gcd(a, b) = sa + tb$  (proved using induction on the value of  $b$  using the Chinese Remainder Theorem). There are known algorithms for finding such integers. This can be built up to determine that given integers  $a_0, \dots, a_n$ , there are integers  $t_0, \dots, t_n$  such that  $gcd^n = \sum_{i=0}^n t_i a_i$ .

**Theorem 3.4.5** *Let  $t_0, \dots, t_n$  be integers such that  $gcd^n = \sum_{i=0}^n t_i a_i$ . Then one solution*

*to  $\sum_{i=0}^n a_i x_i = b$  is*

$$x_i = \frac{t_i b}{gcd^n}$$

Proof: Substituting this solution into the original equation, we get

$$\sum_{i=0}^n \frac{a_i t_i b}{gcd^n} = b$$

We can simplify this to get

$$\frac{b}{gcd^n} \sum_{i=0}^n a_i t_i = b.$$

Since we have  $gcd^n = \sum_{i=0}^n t_i a_i$ , we get  $b = b$ . □

Integers	$n, m, k \in I$
Arrays	$a \in A$
Vars	$x, y \in X$
Procedures	$f \in F$
Exprs	$e \in E ::= a \mid x \mid x(e, e) \mid f(\bar{e}) \mid e + e \mid e * e$
Statements	$c \in C ::= x := e$
Proc Bodies	$b \in B ::= \text{stop} \mid c; b$
Proc Defs	$p \in P ::= \text{fn } x := f(\bar{x}) b$

Figure 3.5 : CORE-MATLAB syntax.

Solving the equations is  $O(n^2)$ , since the number of equations is the constant number of variables involved at each statement times the number of statements, and after each equation is solved, all equations must be updated. The solving process happens once for each inferred clique. Since we proved that under practical conditions there are a constant number of cliques, the total process is  $O(n^2)$ .

## 3.5 Formal Description

To analyze the problem of inferring array sizes, we formalize this problem over a minimal core calculus for MATLAB that we call CORE-MATLAB. All valid programs in CORE-MATLAB are valid MATLAB programs. In Section 3.4, we described type inference informally over a larger subset of the constructs available in MATLAB.

We first describe the subset of MATLAB over which we infer types. We then give the type system for our problem and formalize the solution we informally described in the previous section. We show that our solution is sound and complete with respect to the type system.

### 3.5.1 CORE-MATLAB

Figure 3.5 gives the syntax for CORE-MATLAB. CORE-MATLAB does not include higher-order functions. Many MATLAB features that have little bearing on the ar-



ray size-inference algorithm are omitted in CORE-MATLAB. For example, in CORE-MATLAB, we assume that each procedure returns a single value and that subscripted array accesses only access single elements of an array. Because scalar types are not relevant to array-size inference, we stipulate that all scalars are integers and will not infer primitive types, which would be necessary to properly handle non-integer numbers. For simplicity, we require that all arrays are two-dimensional.

One feature of MATLAB that is relevant to array-size inference that we do not formalize is control flow such as conditional branches, for-loops, and recursion. Although these features are handled in our implementation, we leave a formal analysis of their bearing on array-size inference as future work. The problem is interesting even in the absence of control flow since the overloaded operations are a form of control flow as they require dynamic dispatch based on the input types.

We use “:=” to represent assignment in the formalization to distinguish from equality, although this is not MATLAB syntax. CORE-MATLAB does not support direct assignment to a section of an array. We choose not to support this because we are operating on an SSA form that creates a new variable whenever a section of an array is assigned and copies the new array into the new variable. In the actual implementation, we handle assigning to array sections directly.

A CORE-MATLAB procedure consists of a single output parameter, the procedure name, a set of input parameters, and a procedure body. We use the procedure symbol loosely in that sometimes  $P$  refers to the body of the procedure  $P$ . The procedure body is a series of statements ending in `stop`. Each statement assigns an expression on the right-hand side to a variable. Expressions can be array constants, other variables, array sections, or procedure applications. Included in CORE-MATLAB are two primitive operations,  $+$  and  $*$ . While the type-inference algorithm does not distinguish between primitive operations and procedure calls, we include these primitive operations to show the types of errors that can be caught from static analysis, namely, that certain size constraints hold on inputs to the operations and procedure calls.

The primitive operations also give us starting operations from which all other library procedures may be built.

**Operational Semantics** The evaluation rules, as shown in Figure 3.6, describe the CORE-MATLAB semantics. We evaluate these rules over procedure bodies with a given set of inputs to the procedure.

The set of stores holds all the variable assignments seen thus far in the program. Addition and multiplication are over matrices.

[VAR]: Access to a variable returns the value of the variable if the variable and its value are included in the store.

[E-VAR]: If the variable is not part of the store, access to the variable results in error (*i.e.*, trying to access a variable before it has been defined).

[APP]: The result of function application is the result of evaluating the body of the function with formal parameters replaced by actual parameter values.

[E1-APP]: If the number of actual parameters does not match the number of formal parameters the program results in an error.

[E2-APP]: If the function is not defined, the program results in an error.

[ELEM]: A subscripted variable results in the element represented by the evaluation of the subscripts of the matrix corresponding to the variable as defined in the store.

[E-ELEM]: If the variable is not stored or the array subscripts are not scalars, the program results in an error.

[ADD]: The + operator results in matrix addition.

[E-ADD]: If the sizes of the two matrices do not match, the program results in an error.

$$\begin{array}{c}
\text{Stores } \Sigma \in ST ::= [] \mid x := a, \Sigma \\
\text{Sizeof}(A^{n \times m}) = \langle n, m \rangle \\
\text{Add}(a_1, a_2) = a, \text{ where } a \text{ is result of matrix addition on } a_1 \text{ and } a_2. \\
\text{Mult}(a_1, a_2) = a, \text{ where } a \text{ is result of matrix multiply on } a_1 \text{ and } a_2. \\
\frac{\{x := a\} \in \Sigma}{\Sigma, x \mapsto a} [\text{VAR}] \quad \frac{\{x := a\} \notin \Sigma}{\Sigma, x \mapsto \text{error}} [\text{E-VAR}] \\
\frac{\{ \Sigma, e_i \mapsto a_i \}^{i \in \{1, \dots, n\}} \quad \text{Def}(f) = \text{fn } x_0 := f(x_1, \dots, x_n) b \quad \Sigma, [x_1 \mapsto a_1, \dots, x_n \mapsto a_n] b \mapsto a}{\Sigma, f(e_1, \dots, e_n) \mapsto a} [\text{APP}] \\
\frac{\text{Def}(f) = \text{fn } x_0 := f(x_1, \dots, x_m) b \quad m \neq n}{\Sigma, f(e_1, \dots, e_n) \mapsto \text{error}} [\text{E1-APP}] \quad \frac{f \notin \text{Def}}{\Sigma, f(e_1, \dots, e_n) \mapsto \text{error}} [\text{E2-APP}] \\
\frac{\Sigma, e_1 \mapsto a_1 \quad \Sigma, e_2 \mapsto a_2 \quad \Sigma(x) = a \quad \text{Sizeof}(a_1) = \langle 1, 1 \rangle \quad \text{Sizeof}(a_2) = \langle 1, 1 \rangle \quad \text{Sizeof}(a) = \langle n_1, n_2 \rangle \quad a_1 \leq n_1 \quad a_2 \leq n_2}{\Sigma, x(e_1, e_2) \mapsto a(a_1, a_2)} [\text{ELEM}] \\
\frac{\Sigma, e_1 \mapsto a_1 \quad \Sigma, e_2 \mapsto a_2 \quad \{x := a\} \notin \Sigma \text{ or } \text{Sizeof}(a_1) \neq \langle 1, 1 \rangle \text{ or } \text{Sizeof}(a_2) \neq \langle 1, 1 \rangle}{\Sigma, x(e_1, e_2) \mapsto \text{error}} [\text{E-ELEM}] \\
\frac{\Sigma, e_1 \mapsto a_1 \quad \Sigma, e_2 \mapsto a_2 \quad \text{Sizeof}(a_1) = \text{Sizeof}(a_2)}{\Sigma, e_1 + e_2 \mapsto \text{Add}(a_1, a_2)} [\text{ADD}] \quad \frac{\Sigma, e_1 \mapsto a_1 \quad \Sigma, e_2 \mapsto a_2 \quad \text{Sizeof}(a_1) \neq \text{Sizeof}(a_2)}{\Sigma, e_1 + e_2 \mapsto \text{error}} [\text{E-ADD}] \\
\frac{\Sigma, e_1 \mapsto a_1 \quad \Sigma, e_2 \mapsto a_2 \quad \text{Sizeof}(a_1) = \langle n_1, n_2 \rangle, \text{Sizeof}(a_2) = \langle m_1, m_2 \rangle \quad n_2 = m_1}{\Sigma, e_1 * e_2 \mapsto \text{Mult}(a_1, a_2)} [\text{MULT}] \\
\frac{\Sigma, e_1 \mapsto a_1 \quad \Sigma, e_2 \mapsto a_2 \quad \text{Sizeof}(v_1) = \langle n_1, n_2 \rangle, \text{Sizeof}(v_2) = \langle m_1, m_2 \rangle \quad n_2 \neq m_1}{\Sigma, e_1 + e_2 \mapsto \text{error}} [\text{E-MULT}] \\
\frac{\Sigma, c \mapsto \Sigma' \quad \Sigma', b \mapsto \Sigma''}{\Sigma, c; b \mapsto \Sigma''} [\text{BODY}] \quad \frac{\Sigma, e \mapsto a}{\Sigma, x := e \mapsto \Sigma + \{x := a\}} [\text{STMT}] \quad \frac{}{\Sigma, \text{stop} \mapsto \Sigma} [\text{STOP}]
\end{array}$$

Figure 3.6 : CORE-MATLAB operational semantics.

[MULT]: The \* operator results in matrix multiplication.

[E-MULT]: If the size of the second dimension of the first matrix does not match the size of the first dimension of the second matrix, the program results in an error.

[BODY]: The body is evaluated a single statement at the time from first to last.

[STMT]: Each statement is evaluated by evaluating the expression on the right-hand side and storing the resulting value with the variable on the left-hand side.

[STOP]: The program terminates when stop is reached.

**Type System** We first define the type language used for the size-inference problem shown in Figure 3.7. In the type language, a procedure type is a sequence of mutually exclusive, jointly exhaustive clauses, where each clause denotes a single configuration of types over the procedure parameters. We call these types *mutually exclusive types*. These types are necessary both to define the smallest number of variants and for efficient compilation as we showed in Section 3.4. Procedure types are used in the constraint system when substituting parameter types with argument types at a particular call site. Their non-standard formulation as a sequence of clauses allows us to simplify the description of the constraint system.

The clauses represent each possible type assignment over the inputs and outputs of the procedure. The clauses should be mutually exclusive and jointly exhaustive. That is, no two clauses can hold true at the same time, and the clauses represent every possible type assignment to the parameters of the procedure. The variable types used in the clauses consist of a pair of linear expressions over the  $\$$ -variables representing the size in each dimension.

Figure 3.8 describes the simple type system for the size-inference problem that we impose on CORE-MATLAB. The procedure type as described in Section 3.3 has at most

$$\begin{array}{ll}
\text{\$-Vars} & d \in D ::= \$n \\
\text{\$-Exprs} & s \in S ::= n * d \mid n \mid n * d + s \\
\text{Var Types} & t \in T ::= \langle s_1, s_2 \rangle \\
\text{Clause} & cl \in CL ::= cl \wedge \sigma^x = t \mid \wedge \sigma^x = t \\
\text{Proc Types} & r \in R ::= r \text{ XOR } cl \mid cl
\end{array}$$

Figure 3.7 : Type language.

one clause that represents each scalar/non-scalar assignment to the parameters. The  $\text{\$}$ -variables handle the additional constraints of how the sizes of the parameters relate to each other. To determine that the types of the arguments at an application site are represented in the procedure type by a clause, we define a notion of substitution. If the variable types of the parameters in one of the clauses from the procedure type can be validly substituted by the variable types of the arguments, then the application is well-typed.

Substitution maps  $\text{\$}$ -variables to linear expressions of  $\text{\$}$ -variables. Substitution must not map non-scalar types to scalar types. Also, substitution should recognize that linear expressions are commutative, distributive and associative. Therefore, to ensure, for example, that

$$\mu_S(\$1) = 3 * \$5 + 2 * \$7$$

and

$$\mu_S(\$1) = 1 * \$5 + 2 * \$7 + 2 * \$5$$

both represent the same substitution, we will *normalize* the substitution by introducing a canonical ordering. We sort the terms of a linear combination in the substitution into ascending order of  $\text{\$}$ -var, and add the coefficients of the  $\text{\$}$ -variables within the expression. We call this canonical ordering function *norm*.

These constraints are represented in the definition of a substitution, which we define in three parts.

1. Substitution of  $\text{\$}$ -variables for linear expressions of  $\text{\$}$ -variables:

Type Table	$Type : F \rightarrow R$ (contains function type- for all called procedures).
Def Table	$Def : F \rightarrow P$ (contains function definitions for all called procedures).
Environ	$\Gamma \in G ::= [] \mid x : t \wedge \Gamma$
TSub	$\mu_T : T \rightarrow T$
SSub	$\mu_S : S \rightarrow S$
DSub	$\mu_D : D \rightarrow S$

$$\begin{aligned} \mu_T(\langle s_1, s_2 \rangle) &= \langle \mu_S(s_1), \mu_S(s_2) \rangle \quad s.t. \quad \langle \mu_S(s_1), \mu_S(s_2) \rangle = \langle 1, 1 \rangle \Rightarrow \langle s_1, s_2 \rangle = \langle 1, 1 \rangle \\ \mu_S(k_1 d_1 + \dots + k_n d_n) &= \text{norm}(k_1 \mu_D(d_1) + \dots + k_n \mu_D(d_n)) \\ \mu_D(d) &= s \end{aligned}$$

Expression typing rules:

$$\begin{aligned} &\frac{}{\Gamma \vdash A^{n \times m} : \langle n, m \rangle} [\text{VALUE}] \\ &\frac{\Gamma(x) = t}{\Gamma \vdash x : t} [\text{VAR}] \\ &\frac{\Gamma \vdash e_1 : \langle 1, 1 \rangle \quad \Gamma \vdash e_2 : \langle 1, 1 \rangle}{\Gamma \vdash x(e_1, e_2) : \langle 1, 1 \rangle} [\text{ELEM}] \\ &\frac{\Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 + e_2 : t} [\text{ADD}] \\ &\frac{\Gamma \vdash e_1 : \langle s_1, s_3 \rangle \quad \Gamma \vdash e_2 : \langle s_3, s_2 \rangle}{\Gamma \vdash e_1 * e_2 : \langle s_1, s_2 \rangle} [\text{MULT}] \\ &\frac{\begin{array}{l} Def(f) = \text{fn } x_0 := f(x_1, \dots, x_n) \ b \\ \sigma^{x_0} = t'_0 \wedge \dots \wedge \sigma^{x_n} = t'_n \in Type(f) \\ \Gamma \vdash e_1 : t_1 \dots \Gamma \vdash e_n : t_n \\ \exists \mu_T \ s.t. \ \mu_T(t'_i) = t_i, \ i = 0, \dots, n \end{array}}{\Gamma \vdash f(e_1, \dots, e_n) : t_0} [\text{APP}] \end{aligned}$$

Statement typing rules:

$$\frac{\Gamma(x) = t \quad \Gamma \vdash e : t}{\Gamma \vdash x := e} [\text{STMT}] \quad \frac{\Gamma \vdash c \quad \Gamma \vdash b}{\Gamma \vdash c; b} [\text{BODY}] \quad \frac{}{\Gamma \vdash \text{stop}} [\text{STOP}]$$

Figure 3.8 : Type system for size problem.

$$\mu_D(d) = s$$

2. Substitution of linear expressions of  $\$$ -variables to linear expressions of  $\$$ -variable:

$$\mu_S(k_1d_1 + \dots + k_nd_n) = \text{norm}(k_1\mu_D(d_1) + \dots + k_n\mu_D(d_n))$$

3. Substitution of variable types to variable types:

$$\mu_T(\langle s_1, s_2 \rangle) = \langle \mu_S(s_1), \mu_S(s_2) \rangle$$

$$s.t. \langle \mu_S(s_1), \mu_S(s_2) \rangle = \langle 1, 1 \rangle \Rightarrow \langle s_1, s_2 \rangle = \langle 1, 1 \rangle$$

It is assumed that we are given tables, *Type* and *Def* that map procedure names of the called procedures to the corresponding procedure types and definitions respectively for all the called procedures.

[VALUE]: A matrix value has a type that is the size of the matrix in each dimension.

[ELEM]: An element of an array is a scalar.

[MULT]: The type of the result of multiplication is the size of the first dimension of the first matrix and the size of the second dimension of the second matrix.

[APP]: To get the type of the result of function application, first the procedure type from the type table must be determined. Then the clause with types of the formal parameters that match the types of the actual parameters is found. The type of the result of the procedure in this clause is the type of the result of function application.

Type inference over a procedure  $P$  is described via the notion of a *Principal* of  $P$ , which is a set of distinct type environments  $\Gamma_1, \dots, \Gamma_n$  such that each  $\Gamma_i$  is a type environment corresponding to a particular type configuration of  $P$

$$\text{Princ}(P) = \{\Gamma \mid \Gamma \vdash P \text{ } \text{Vars}(P) = \text{dom}(\Gamma)\}$$

To have a valid type derivation of a procedure  $P$ , one must assume that the procedure type for any procedure  $f$  called in  $P$  must give only the input and output type configurations that allow  $f$  to be well-typed. Since the type inference process infers these entries, we show that, given correct entries for  $Type$  on the primitive operations, the correct procedure type is inferred for any procedure that is built up from the primitive operations. Therefore, any entry to  $Type$  entered from results of type inference is correct.

To show all of this, we first prove that the type system we present is type safe with respect to CORE-MATLAB, given that  $Type$  is correct. We then show that we can infer exactly the set of type environments,  $\Gamma$ , that allow for a type derivation over a procedure,  $P$ . It is trivial to show that by inferring  $Principal$  for  $P$ , we can extend  $Type$  to include a new entry for  $P$ , that takes only the information from the input and output parameters from each  $\Gamma$  in  $Principal$ , and combines them using  $\vee$  to produce the procedure type. Since  $Type(P)$  is based on a correct  $Principal$  for  $P$ , the new  $Type$  is still correct.

**Safety** We define the following judgment to relate type environments to stores in the evaluation rules.

$$\frac{\forall \{x := a\} \in \Sigma \quad \Gamma \vdash x : t \quad \cdot \vdash a : t}{\Gamma \vdash \Sigma}$$

Appendix A provides a proof for the following type safety theorem.

**Theorem 3.5.1** *Type Safety:*

*Given  $Type$ , if  $\Gamma \vdash \Sigma, \Gamma \vdash e : t$  and  $\Sigma, e \hookrightarrow a$  then  $\Gamma \vdash a : t$ . Also, if  $\Gamma \vdash \Sigma, \Gamma \vdash P$ , and  $\Sigma, P \hookrightarrow \Sigma'$ , then  $\Gamma \vdash \Sigma'$ .*

Note that this theorem ensures that no well-typed procedure results in an error.



### 3.5.2 Deriving the Principal Type

The typing rules only state that a procedure is correct with respect to a given *Principal*. What they do not state is how we derive *Principal* for the procedure. We now formally describe the process by which we derive all valid  $\Gamma$ 's, which we will informally describe in Section 3.4.

*Principal* can be derived in two steps. First, the procedure constraint must be formed from the statement constraints. Then the procedure constraint must be solved. We show that these two steps are sufficient to find *Principal*.

**Building the Procedure Constraint** Before procedure constraints can be built, the procedure must be transformed so that each statement contains only one operation or procedure call, since constraints are formed over statements and not operations. This normalization is described in Figure A.2 in the Appendix, as well as a proof that normalization preserves types. We define the function *nrm* as the function that normalizes procedures.

As shown in Figure 3.9, statement constraints are formed from procedure types, with the parameters replaced by the arguments. The  $\$$ -variables are replaced by  $\$$ -variables that are fresh in the procedure constraint to isolate the statement constraints from each other. If one of the actual parameters involved in the statement is a subscripted array access, an extra atom stating that the argument must be a scalar is appended to the statement constraint. This has the effect of negating any clause that treats that argument as a non-scalar. In the actual implementation, the compiler just eliminates these clauses.

**Solution for the Procedure Constraint** The procedure constraint merely states what properties must hold over the variable types for the procedure to be valid. It does not give the possible type configurations, or  $\Gamma$ , necessary for variant generation. The procedure constraint must be solved to determine all possible type configurations.

Proc Constraints  $pc \in PC ::= pc \wedge sc \mid \epsilon$   
 Stmt Constraints  $sc \in SC ::= sc \vee cl \mid cl$   
 Expr Constraints  $ec \in EC ::= \sigma^e = t$   
 Expressions  $e \in E ::= a \mid x \mid x(e, e)$   
 $\$-vars(r) = \bar{d}$ , where  $\bar{d}$  is the set of  $\$$ -vars used in  $r$ .

$$\begin{array}{c}
 \frac{}{\sigma^{A^{n,m}} = \langle n, m \rangle} [\text{VALUE}] \\
 \\
 \frac{}{x : \epsilon} [\text{VAR}] \frac{}{x(e_1, e_2) : \sigma^{x(e_1, e_2)} = \langle 1, 1 \rangle \wedge \sigma^{e_2} = \langle 1, 1 \rangle \wedge \sigma^{e_1} = \langle 1, 1 \rangle} [\text{ELEM}] \\
 \\
 \frac{e_1 : ec_1, \dots, e_n : ec_n \quad \text{Type}(f) = r \quad \text{Def}(f) = \text{fn } x_0 := f(x_1, \dots, x_n) \quad P \quad \$-vars(r) = \bar{d} \quad \bar{d}' \text{ fresh} \quad |\bar{d}'| = |\bar{d}|}{y = f(e_1, \dots, e_n) : \quad ([x_0 \mapsto y, x_i \mapsto e_i][\bar{d} \mapsto \bar{d}']r) \wedge ec_1 \wedge \dots \wedge ec_n} [\text{STMT}] \\
 \\
 \frac{}{\text{stop} : \epsilon} [\text{STOP}] \frac{c : sc \quad P : pc}{c; P : sc \wedge pc} [\text{PROC}]
 \end{array}$$

Figure 3.9 : Simple constraint system for size problem.

We describe the process for solving the constraints in Section 3.4. We formalize the solution to the constraint system in Figure 3.10.

A  $\Gamma$  satisfies the constraints if for all the program constraints, there exists a valid mapping,  $\mu_T$ , from  $\$$ -variables to linear expressions over  $\$$ -variables that match the variable types in  $\Gamma$ .

**Soundness and Completeness of Solution** We show that our solution is both sound and complete with respect to the type system given in Figure 3.8. The proofs are given in Appendix A.

**Theorem 3.5.2** *Soundness: Given Type, Def, and P, if  $\mu_T, \Gamma \vdash pc$  and  $nrm[P] : pc$  then  $\Gamma \vdash P$ .*

**Theorem 3.5.3** *Completeness: Given Type, Def, and P, if  $\Gamma \not\vdash P$  then there does*

$$\begin{array}{c}
\frac{cl = \sigma^x = t \wedge cl' \quad \Gamma(x) = \mu_T(t) \quad \mu_T, \Gamma \vdash cl'}{\mu_T, \Gamma \vdash cl} [\text{CLAUSE}] \\
\\
\frac{cl = \sigma^x = t \quad \Gamma(x) = \mu_T(t)}{\mu_T, \Gamma \vdash cl} [\text{CLAUSE2}] \\
\\
\frac{sc = cl \vee sc' \quad \mu_T, \Gamma \vdash cl \vee \mu_T, \Gamma \vdash sc'}{\mu_T, \Gamma \vdash sc} [\text{STMT-CONSTR}] \\
\\
\frac{sc = cl \quad \mu_T, \Gamma \vdash cl}{\mu_T, \Gamma \vdash sc} [\text{STMT-CONSTR2}] \\
\\
\frac{pc = sc \wedge pc' \quad \mu_T, \Gamma \vdash sc \quad \mu_T, \Gamma \vdash pc'}{\mu_T, \Gamma \vdash pc} [\text{PROC-CONSTR}] \\
\\
\frac{pc = \epsilon}{\mu_T, \Gamma \vdash pc} [\text{PROC-CONSTR2}]
\end{array}$$

Figure 3.10 : Well-formed type configurations.

*not exist a  $\mu_T$  such that  $\mu_T, \Gamma \vdash pc$ , where  $\text{norm}[P] : pc$ .*

## Chapter 4

### Extensions to Type-Inference Algorithm

Chapter 3 described the basic size inference algorithm over single MATLAB procedures without control flow. This chapter describes extensions to the basic algorithm that enable type inference over programs that occur in practice. We first discuss ways to extend type analysis to handle the widely-used program constructs found in MATLAB and other languages such as control flow and access to array sections. We show that annotations from library writers concerning types are easily incorporated into the type inference system to help reduce the number of type configurations inferred. We then discuss interprocedural issues for the type-inference algorithm. While the telescoping-languages strategy does not require whole-program analysis, we still need to consider how type inference is handled across procedures. Finally, we show how type inference is applicable to problems other than the size inference problem that are important for achieving high performance from MATLAB and other languages.

#### 4.1 Handling Control Flow

In Chapter 3, we presented the type-inference algorithm under the assumption that there was no control flow and that all procedures were *type-input-dependent* (i.e., the types of the outputs can be determined directly from the types of the inputs). Of course, many MATLAB procedures do involve control flow constructs, and not all operations in MATLAB are type input-dependent. For some operations, the types of the outputs could depend on the values of the inputs, rather than on the types. This means that an operation could produce multiple output types for a given set of input types. Variables defined by such operations and by the outcome of control flow are

u-vars.  $u$  should still remain small, since few operations are not input-dependent and the amount of control flow is usually limited. Note that the algorithm works without these assumptions, but the complexity is exponential in the worst case.

Loops as well as branch statements involve control-flow constructs. In SSA,  $\phi$ -nodes represent merge points in the control flow. For the type inference problem, the  $\phi$ -nodes represent the “meet” operation over the types of the variables involved along each path. This section describes two strategies for extending the type-inference algorithm to handle control flow. Our original strategy only infers types correctly under certain, albeit likely, assumptions, and is not precise (*i.e.*, may infer more type configurations than are valid). The second strategy is more precise and will correctly handle all valid procedures.

#### 4.1.1 Original Strategy

Our original strategy is a straightforward extension of the type-inference algorithm described in the previous chapter. A single constraint graph is generated to represent constraints over all the basic blocks in the control flow graph. The constraints for the blocks are listed in reverse-post order in the constraint graph.

The constraint graph does not represent the  $\phi$ -statements directly. However, the introduction of a new variable defined by the  $\phi$ -node creates an u-var since the types of the u-var in this strategy cannot necessarily be determined by the types of the variables defined before its definition.<sup>1</sup> Type inference proceeds as it does for straight-line code.

Although  $u$  could now be as large as the number of statements, in actuality, this number should still remain small, since the amount of control flow rarely reaches this upper bound. Note that the use of the variable is required for the variable to appear in a constraint, and the constraints from the use will help limit the possibility of extra

---

<sup>1</sup>The compiler can get the benefit of working with pruned SSA without requiring the code to be in the pruned form, since if a variable is never used, it never appears in a constraint.

complexity caused by control flow.

There are two disadvantages to the original approach. The first is that the type of the variable defined by the  $\phi$ -node is in no way dependent on the types of the inputs to the  $\phi$ -function. In other words, the type of this variable is only determined from its uses. This can lead to imprecision in the result as well as inefficiencies in the algorithm, since not only is the variable defined in the  $\phi$ -node not constrained by the arguments to the  $\phi$ -node, the arguments are not constrained by subsequent uses.

The other problem with this approach is that because the statements from all of the basic blocks are combined in a single constraint graph, the restrictions on the variable types from statements in the individual paths can restrict the variable types over the whole procedure. For example, a parameter may be used as a scalar along one path, but another path may use the parameter as a non-scalar. The original strategy would result in no solution, when either case is valid depending on the control flow, since MATLAB supports different types for the same variable along different paths. While cases like these are not frequent, to assert that the solution is complete (*i.e.*, covers all possible type configurations) this problem must be addressed.

#### 4.1.2 Complete Strategy

To address the problems with the original strategy, we developed a new strategy for performing type inference in the presence of control flow. In this strategy, type inference is performed over each basic block separately and then the information from each block is merged along the edges of the control-flow graph. This strategy is shown in Figure 4.1.

The blocks in the control-flow graph are visited in reverse post order (*i.e.*, all the predecessors to a block that do not come from a back edge are visited before the block). After the cliques have been found and solved over a particular basic block, the solution is converted into a single statement constraint involving all the variables that have been used or defined on the paths from the entry node that include that

```

inferTypes
input: Function f
output: Type Jump-Function sc
1 cfg = f.getCFG()
2 changed = true
3 while (changed)
4   changed = false
5   for every n in cfg visited in reverse post order
6     oldSC = n.summaryConstraint
7     predConstraint = emptyConstraint
8     predsChanged = false
9     for every cfg node p s.t. p is a predecessor of n
10      predConstraint = merge(predConstraint, p.summaryConstraint)
11      if (p.hasChanged)
12        predsChanged = true
13    if (predsChanged)
14      constraintList = buildConstraintList(n)
15      constraintList.addFrontPhiConstraints(n)
16      constraintList.addFront(predConstraint)
17      graph = buildGraph(constraintList)
18      cliques = findCliques(constraintList)
19      n.summaryConstraint = solveCliques(cliques)
20    if (n.summaryConstraint != oldSC)
21      n.hasChanged = true
22      changed = true
23    else
24      n.hasChanged = false
25 sc = n.summaryConstraint s.t. n is last node in reverse post order

```

Figure 4.1 : Type inference in the presence of control flow.

block. This constraint, which we term the summary constraint, is then prepended to the constraint graphs for each of the successor blocks.

If a block has multiple predecessors, all summary constraints from its predecessors are merged by conjoining them through logical disjunction, and then transforming this new constraint so that the clauses are mutually exclusive. In this way, all possible type configurations of the variables over all the preceding paths are still possible.

Merge also removes from the summary constraints all equations involving variables defined in the successor block, so that types are not over-constrained by previous iterations.

The solution as we have described it thus far handles the second problem from the original strategy of over-constraining the variable types. However, the algorithm still does not solve the first problem of losing information at merge points. To solve this problem, we insert constraints that represent the merge of types at the  $\phi$ -nodes. Clauses in the  $\phi$ -node constraint are associated with a particular predecessor so that  $\phi$ -node clauses are only compatible with the clauses in the summary constraint from the predecessor to the block that correspond to the CFG node with which the  $\phi$ -node clause is associated. This ensures that the constraints at the  $\phi$ -node accurately reflect the flow of types into the block and avoids imprecision.

Since the summary constraints along the paths leading up to the block with the  $\phi$ -node are unioned together, the type of the variable defined by the  $\phi$ -function is constrained to the types of the inputs along the incoming paths. Thus, more precise information is inferred. Note that because type inference infers types of variables from uses as well as the definition, the variables that are the inputs to the  $\phi$ -node will be constrained by the type inferred for the result of the  $\phi$ -node.

Because loop bodies must be evaluated before type inference has been performed on all of their predecessors, this process must be repeated until a fixed point is reached. While type inference does not meet the rapid condition [54], it is guaranteed to halt. Changes in variable types that occur from iterating over a back edge are all dependent on the changes to the types of the inputs to the  $\phi$ -function. This is the purpose of the  $\phi$ -node in the type-inference algorithm - to represent any possible changes in type that occur through merging the types at any node. Since the number of types any variable can take is bounded by the size of the type lattice (for size, this is just 2 - scalar and non-scalar), and since the types can only grow around back edges ( $\phi$ -nodes represent a merging of types), the number of iterations is bounded by the number of



$\phi$ -nodes at back edges times the number of elements in the lattice, which is constant. However, in practice, the algorithm converges much faster, since the only case where this can occur is if there is a trail of operations that grow arrays from scalars based on the sizes of the other variables.

### 4.1.3 Complexity

Type inference for each basic block takes time  $O(2^u m^2)$ , where  $m$  is the number of operators in each basic block. This complexity result is from the previous chapter (since none of the previous order requirements have been violated by the extension). If the number of basic blocks is  $b$ , then the total complexity is  $O(2^u m^2 b)$ , assuming no back edges. But since the number of statements in the entire procedure is approximately  $mb$ , the complexity is  $O(2^u mn)$ , where  $m \leq n$ .  $u$  could still be large in the worst case because the procedure may have calls to routines that are not input dependent. Again, in practice the number of u-vars should remain small. Also, because the new strategy is more precise, fewer cliques will be inferred at each step in the clique finding algorithm. However, the presence of back edges causes the complexity to be multiplied in the worst case by the number of  $\phi$ -nodes with inputs defined from a back edge, which in the worst case is  $n^2$ . This upper bound is only reached if there is a chain of resizing operations that convert scalars to arrays based on the sizes of the other variables. Otherwise, the number of iterations is very small.

### 4.1.4 Specialization

Control flow and non input-determined operations may cause multiple cliques that represent the same type configuration over the arguments, but different types for the local variables. The compiler must be able to determine which specialized variant of the general procedure to use at a given call site based on the types of the types of the inputs and outputs. Therefore, separate variants should not be generated with the same types for the inputs and outputs since these variants would be indistinguishable

$$\begin{array}{l}
\mathbf{t} = \mathbf{V}(:, 1:j) * \mathbf{h} \\
\\
\sigma^{\mathbf{t}} = \langle 1, 1 \rangle \quad (\wedge \sigma^{\mathbf{V}(:, 1:j)} = \langle 1, 1 \rangle) \quad \wedge \sigma^{\mathbf{h}} = \langle 1, 1 \rangle \quad \text{XOR} \\
\sigma^{\mathbf{t}} = \langle \$1, \$2 \rangle \quad (\wedge \sigma^{\mathbf{V}(:, 1:j)} = \langle 1, 1 \rangle) \quad \wedge \sigma^{\mathbf{h}} = \langle \$1, \$2 \rangle \quad \wedge (\$1 \neq 1 \vee \$2 \neq 1) \quad \text{XOR} \\
\sigma^{\mathbf{t}} = \langle \$1, j \rangle \quad (\wedge \sigma^{\mathbf{V}(:, 1:j)} = \langle \$1, j \rangle) \quad \wedge \sigma^{\mathbf{h}} = \langle 1, 1 \rangle \quad \wedge (\$1 \neq 1 \vee j \neq 1) \quad \text{XOR} \\
\sigma^{\mathbf{t}} = \langle \$1, \$3 \rangle \quad (\wedge \sigma^{\mathbf{V}(:, 1:j)} = \langle \$1, j \rangle) \quad \wedge \sigma^{\mathbf{h}} = \langle j, \$3 \rangle \quad \wedge ((\$1 \neq 1 \wedge j \neq 1) \vee \\
\quad \quad \quad (\$3 \neq 1 \wedge (\$1 \neq 1 \vee j \neq 1))) \quad \text{XOR} \\
\sigma^{\mathbf{t}} = \langle 1, 1 \rangle \quad (\wedge \sigma^{\mathbf{V}(:, 1:j)} = \langle 1, j \rangle) \quad \wedge \sigma^{\mathbf{h}} = \langle j, 1 \rangle \quad \wedge (j \neq 1)
\end{array}$$

Figure 4.2 : Statement constraints in the presence of array sections.

during variant selection. However, the compiler can generate specialized paths from the control-flow points and operations if it can determine that optimization would be beneficial. For example, it may be beneficial to have separate paths if the outcome of the  $\phi$ -node could either be real or complex. Since this could cause an increase in the size of the code, the compiler must be careful about how many specialized paths are generated. Ultimately, the compiler could allocate the meet of the types to the variable over all paths. We discuss further specialization opportunities in the presence of control flow in the next chapter.

## 4.2 Subscripted Array Accesses

MATLAB has support for using and assigning to sections of arrays. When only a section of an array is accessed in a statement, the size of the whole array is not constrained by the statement. The sizes of the other variables are, however, constrained by the size of the section of the array accessed. Since the values of constants or other program variables determine the size of the section, this information can be very helpful in more accurately inferring sizes.

Subscripted array accesses can also reduce the complexity of the type-inference algorithm if certain nodes in the constraint graph are found to be invalid given information about the size of the piece accessed. For example, if the statement is only

accessing a single element of the array, then that argument or result is a scalar and nodes that constrain the arguments to non-scalars are eliminated.

If the size of the array section is defined in terms of the value of a program variable, the compiler needs to account for the fact that the variable could have a value of one at runtime, making the access scalar. Figure 4.2 illustrates how the constraints are written to handle this situation. The first fields of  $\sigma^v$  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.

The subscripts used to access the array section may also give some information about the size of the whole array. The variable must be at least the size of the value of the largest subscript. If the compiler determines that the subscript accesses elements past the first element, then it can add a constraint that forces the variable to be non-scalar, reducing the number of cliques.

If an array section is assigned to, then the size of that array is the value of the largest subscript in each dimension. This holds true because it is assumed that redefining parts of an array creates an entirely new array in SSA, therefore this assignment to the array section is the only assignment to the array over the whole procedure. The arrays will be remerged during code generation and allocated to the maximum size of all of the pieces.

To take advantage of subscript information when the subscripts are written in terms of procedure variables, the compiler must first perform a variation of constant propagation to determine the maximum value of the variables if possible and, therefore, the maximum array sizes. Note that a maximum size is only needed when there is a subscripted assignment to the array in a loop to avoid reallocation in loops if the array grows. Determining a minimum value of the variable would also be useful to prove that the section of the array is not a scalar.

Subscripted array accesses do not affect intrinsic type inference since all the array elements are defined to have the same intrinsic type. For the pattern inference prob-

```

[o1,o2] = size(i1)

 $\sigma^{\mathbf{o1}} = \langle 1, 1 \rangle \quad \wedge \quad \sigma^{\mathbf{o2}} = \langle 1, 1 \rangle \quad \wedge \quad \sigma^{\mathbf{i1}} = \langle 1, 1 \rangle$ 
 $\sigma^{\mathbf{o1}} = \langle 1, 1 \rangle \quad \wedge \quad \sigma^{\mathbf{o2}} = \langle 1, 1 \rangle \quad \wedge \quad \sigma^{\mathbf{i1}} = \langle \$1, \$2 \rangle \quad \wedge \quad \$1 = \mathbf{o1} \quad \wedge \quad \$2 = \mathbf{o2}$ 
XOR

o1= zeros(i1, i2)

 $\sigma^{\mathbf{o1}} = \langle 1, 1 \rangle \quad \wedge \quad \sigma^{\mathbf{i1}} = \langle 1, 1 \rangle \quad \wedge \quad \sigma^{\mathbf{i2}} = \langle 1, 1 \rangle$ 
 $\sigma^{\mathbf{o1}} = \langle \$1, \$2 \rangle \quad \wedge \quad \sigma^{\mathbf{i1}} = \langle 1, 1 \rangle \quad \wedge \quad \sigma^{\mathbf{i2}} = \langle 1, 1 \rangle \quad \wedge \quad \$1 = \mathbf{i1} \quad \wedge \quad \$2 = \mathbf{i2}$ 
XOR

```

Figure 4.3 : Return type-jump-function for MATLAB operators `size` and `zeros`.

lem, array sections should not constrain the whole array, since the section accessed could affect the pattern in undeterminable ways.

### 4.3 Value Dependent Operators

MATLAB has a number of operators where the sizes of the arguments determine the value of the results or vice versa. For example, `size` takes a matrix as an argument and produces the size in each dimension. Type inference can use these statement to more accurately infer the relationship between the sizes of the different variables. `size` is often used to define local variables to have the same size as the inputs. Another routine that is commonly used in MATLAB is `zeros`. `zeros` is used frequently by MATLAB programmers since it has the effect of preallocating the array. This turns out to be an important optimization to keep the MATLAB performance under control since otherwise if the array grows in the loop, it may need to be reallocated on every iteration.

To take advantage of the information provided by these statements, we must modify the structure of the return type-jump-function to constrain the `$`-variable's values to be the values of the program variables. Figure 4.3 shows the return type-jump-functions for the two operators mentioned above.

Once the  $\$$ -variable's are set in the statement constraints, we can store the information that this  $\$$ -variable has the same value as the program variable. This can be used to allocate the arrays whose size depends on  $n$ . Unfortunately, since we rely on SSA, `zeros` does not provide much new information in the analysis phase, since redefining a section of the array is treated as an entirely different array. We can however use this information to allocate the arrays once they are remerged.

#### 4.4 Using Annotations

One of the key ideas in telescoping languages is allowing the compiler to leverage the knowledge of the library writer through annotations. This information is important to know what cases can and cannot occur in practice, which the compiler alone may not be able to infer. Annotations also give information to the compiler so that it can perform specializations that it would otherwise not know to be legal or useful.

For type-based specialization, the library writer can provide type information on the parameters of the procedure to be analyzed. In the absence of source code for a procedure, these annotations can be transformed into the procedure's return type-jump-function. The compiler treats these annotations as constraints on the procedure header, which corresponds to the zeroth level in the graph. Any cliques occurring in the graph must have part of the user-defined annotations as one of their nodes. Annotations can greatly reduce the number of possible cliques and, therefore, specialized variants, since they give type constraints to which all cliques must adhere. Because of the reduction in the number of cliques at each stage, they can reduce the runtime of the algorithm. If there are no annotations provided, the header is assumed to be unconstrained.

Annotations can also be used to split out important cases to signal the compiler that there separate variants should be generated. If there are two separate clauses in an annotation stating that a matrix can either be sparse or dense, the type-inference algorithm will propagate this separation wherever that matrix is used, so that ulti-

mately there will be separate cliques for the sparse and dense cases. Otherwise, with no annotation, type inference would have assumed that the most general type (dense) would suffice.

Type annotations are not type declarations, but merely provide hints to the compiler for optimization purposes. The ability to incorporate type annotations is a feature in our type-inference algorithm. Traditional type inference for the purpose of static checking would assume there was a programming error if the type were used in a more specific way than the type that was declared.

Annotations on the types of the parameter are also useful for variant generation. They give hints to the compiler that generating a particular variant will be important in practice - something the compiler cannot necessarily infer on its own. We discuss annotations as they apply to variant selection and generation more in the next chapter.

## 4.5 Interprocedural Type Inference

Because the return type-jump-function for a procedure stores all the type information required for any call to a procedure, when inferring types for a calling procedure, type inference does not need to be re-performed for the called procedure. To ensure that the return type-jump-function is available to the calling procedure, the call graph must be traversed in post order (*i.e.*, all of a node's successors must be visited before visiting the node itself). Cycles in the call graph are a special case. Also, source code, and therefore return type-jump-functions, may not be available for all called procedures.

### 4.5.1 Function Parameters

MATLAB allows the user to pass in functions along with a variable number of arguments as parameters to a procedure. The procedure is then evaluated with the appropriate number of arguments using the `feval` operation. From the statement

that calls `feval`, the parameters passed to the function parameter can be known. The type-inference algorithm can constrain the types of these parameters based on subsequent uses as well as determine the number of parameters. Thus, the function parameter is constrained so that parameter numbers match the number inferred and the parameter types match the type information found. Unfortunately, the parameters to the passed-in function are rarely used outside of the `feval` statement, in which case, type inference must rely entirely on library-writer annotations.

#### 4.5.2 Missing Return Type-Jump-Functions

When the algorithm encounters a procedure call or a built-in operation, it looks in the database for the appropriate return type-jump-function to build the constraints at that statement. If the compiler encounters a procedure call for which there is no return type-jump-function, it simply considers variables as unconstrained by that statement. This degrades the precision of the algorithm, as it may infer more type configurations than are legal. However, all legal configurations will still be inferred. The library writer may also supply an annotation for the procedure that could serve as its return type-jump-function, which reduces analysis time and improves precision.

#### 4.5.3 Recursion

In the case of a cycle in the call graph, or recursion, precision can be enhanced by iterating over the cycle until a fixed-point is reached.

Figure 4.4 describes the algorithm for interprocedural type inference. First, strongly connected components in the call graph, or *scc*'s are found using Tarjan's algorithm [76]. If there is no recursion, each function is an *scc*. These *scc*'s are then traversed in post order over the call graph (*i.e.*, if there is an edge from one *scc* to another, the second must be visited before the first).

First, header constraints for each function in the *scc* are found. If there is no annotation available for that function then the header constraint represents all possible

```

Interproc
input: call graph G
output: type jump-function for each function in G
1 sccList=findSccs(G)
2 for every scc s in sccList visited in post-order
3   for every fn f in s
4     f.headerConstraint=buildHeaderConstraint(f)
5     f.cliques=inferTypes(f)
6   changed=true
7   while (changed)
8     changed=false
9     for every function f in s
10      for every node n in f.headerConstraint
11        partOfClique=false
12        for every clique c in f.cliques
13          if (n in c)
14            partOfClique=true
15        if(!partOfClique)
16          removeNodeFromHeaderForFn (n,f)
17          for every function cf in s s.t. cf calls f
18            for every node m in cf.constraintGraph s.t.
19              m corresponds to n at call site
20              for every clique c in cf.cliques s.t. m in c
21                changed=true
22                removeCliqueFromFn(c, cf)
23 for every fn f in s
24   for every clique c in f.cliques
25     solution = solve(c)
26     addSolToTJF(solution,f.typeJumpFunction)

```

Figure 4.4 : Interprocedural algorithm.

type configurations over the inputs and outputs. If there is a call to a function in the scc, the statement constraint for this call site is also unconstrained (unless there was an annotation provided). Then, the cliques are found in the constraint graph for the entire function including the header. Note that after cliques have been formed in the initialization step, the cliques already include valid type configurations over



the procedures. The iterative step, which we describe next is designed to refine the configurations further, resulting in more precise types.

If a node at the zeroth level in the constraint graph (*i.e.*, the level corresponding to the header constraint) for one of the functions in the scc is not a part of any clique, the node can be removed from the graph as well as the corresponding clause in the return type-jump-function. This means that the call sites for this function can be updated as well by removing the node that corresponds to the invalid clause. If this node belonged to some cliques in the constraint graph for the calling function, these cliques are no longer valid and can be removed from this graph. This in turn may mean that more nodes at the zeroth level for the calling function do not belong to any cliques, etc. The algorithm iterates until a fixed point is reached.

A fixed-point can be reached in a constant number of iterations for each procedure involved. At each iteration, the type-inference algorithm tries to reduce the number of clauses in the return type-jump-function for the procedure (initially the return type-jump-function contains all possible type configurations over the input and output arguments). Since there are only a constant number of clauses, this number can only be reduced a constant number of times.

## 4.6 Other Type Problems

The type-inference algorithm can be applied to all of the type problems discussed in Section 3.2.2. The compiler infers each element of  $\mathcal{T}$  in a separate pass and then takes the cross product of the different type configurations to determine which variants are necessary. It can handle types separately since, except for dimensionality and size, the types are independent of each other, although knowing that a variable is scalar makes pattern inference unnecessary.

### 4.6.1 Handling Multiple Dimensions

The type-inference algorithm described here can easily be extended to handle inferring sizes over multiple dimensions by simply increasing the number of fields in  $\sigma$ .

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

Inferring sizes of arrays with dimension over two should not increase the complexity, since there are still only two possible types for each variable - scalar or non-scalar. If the number of explicitly accessed dimensions grows arbitrarily, this could increase the time to build the constraints, build the graph, and solve the constraints. However, it is unlikely that the procedure will explicitly access an arbitrary number of dimensions.

Because MATLAB was designed around matrices, we assume a minimum dimensionality of two.

### 4.6.2 Pattern and Intrinsic Type Problems

The problems of inferring intrinsic types and patterns differ from inferring size in that the algorithm only operates on finite lattices. Therefore, the constraints are formulated differently. The constraints must restrict variables to a range of types on the respective lattice. Using ranges represents the fact that the specialized variants replacing the call can accept inputs of types that are lower in the lattice than the declared input types. For example, an input argument that is defined as type `real`

```

o=mean(i)

(real ≤ τo ≤ real) ∧ (⊥ ≤ τi ≤ real)   XOR
(comp ≤ τo ≤ comp) ∧ (comp ≤ τi ≤ comp)

```

Figure 4.5 : Intrinsic type constraints on mean operation.

could actually be of type `int` when called. Using ranges also reduces the number of cliques, since each node can represent multiple possibilities.

The constraints for intrinsic types on the *mean* operation are shown in Figure 4.6.2.

Two constraint clauses are not compatible if a variable appears in both clauses and the corresponding ranges do not intersect. The compiler still needs mutual exclusivity for the algorithm to run efficiently. Also, since maintaining the type-input-dependence property is important in reducing the complexity, when possible, the constraints are formulated so that the same input configuration should give only one configuration over the output types. Once the compiler has found the cliques, solving the equations corresponds to taking the intersection of all ranges for each variable over the clique.

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

The number of steps for finding cliques for these problems is bounded by  $l^u n^2$  steps, where  $l$  is the number of lattice elements, which should be bounded by a small constant. Therefore, the overall time complexity is still  $O(n^2)$  if  $l^u$  is bounded by a constant, which is true when operations are primarily type-input-dependent.

Because array patterns do not behave predictably in many operations, the compiler must rely more heavily on user-defined annotations from the library writer to isolate important patterns for the routines. Currently, type inference by default

assumes that all arrays are dense. If an annotation states that other shapes are important, type inference will propagate this information through the procedure to generate separate cliques and thus separate variants.

TeleGen allows the library writer to extend the type inference problem with new type problems not handled by size, shape and intrinsic type. New problems, like intrinsic type and shape, must work on a single finite lattice. The library writer must provide a new base lattice for the new type problem, as well as methods for determining if clauses contradict and for solving the equations from the cliques. At Rice, two projects have extended the type-inference algorithm to solve two new problems - data distributions and object-oriented types. Both of these problems were easy to implement due to the flexibility of the lattices. We discuss these problems in more detail in Sections 4.7 and 4.8.

The type-inference algorithm can be applied to type problems that do not conform to this standard as well (*e.g.*, unit and dimension types), although extending the algorithm to handle these problems is not as straightforward.

The next section discusses how the type-inference algorithm can be applied to user-defined types. New constraints are introduced that could also improve the precision of the intrinsic type and pattern solution with very little added complexity.

## 4.7 Distribution Inference

TeleGen is also able to compile libraries written in Matlab D [40, 39, 38]. Matlab D extends MATLAB with distribution functions. A preliminary Matlab D compiler has been implemented in TeleGen. For Matlab D, the compiler automatically parallelizes the code by translating the MATLAB code into Fortran with communication and partitioned arrays.

Distribution analysis solves the problem of finding distributions for the arrays used in the procedure. We derived a simple distribution analysis that is a straightforward extension of the lattice-based type-inference algorithm. The distribution functions

serve the purpose of annotations. This algorithm is sufficient to find the most appropriate data distributions under the assumption that no redistribution is required or beneficial.

## 4.8 Type Inference for Object-Oriented Languages

In this section, we describe methods to extend type inference to infer user-defined types. We use Python as an example language.

Recently, Python has had growing success among the scientific community. Python is a dynamically-typed, object-oriented language. To support object-oriented features in the telescoping compiler, the type-inference algorithm must be extended to handle user-defined types.

By performing type inference for Python, the compiler, in many cases, will be able to eliminate dynamic dispatch. Dynamic dispatch has traditionally caused a degradation in performance. Also, further cross-method optimizations are possible if the result of dispatch is known statically.

The type-inference strategy for user-defined types is similar to the previously described strategy for inferring primitive types. However, for intrinsic types, the type inference engine only needs to handle a bounded number of types. Also, in an object-oriented world, the type lattice can be extended at any point, including after type inference has been performed.

We first discuss the type-inference strategy for object-oriented types and then describe how to handle the problem of extending the type lattice.

A project at Rice plans to use type inference to uncover more opportunities for object inlining, which has been shown to be an important optimization for making Java more efficient [13]. For this project, preliminary object-oriented type inference has been implemented for Java by a straightforward extension to our type-inference algorithm. This implementation will need to be modified to handle the problems discussed in this section. Note that the strategies applied to Python can be applied

```

class Shape :
    origin = (0,0)
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
    def move(self, delta_x, delta_y):
        self.x += delta_x
        self.y += delta_y
        return self
class Circle(Shape):
    def __init__(self,x=0,y=0,radius=1):
        Shape.__init__(self, x, y)
        self.radius = radius
    def area(self):
        return self.radius * self.radius* 3.14159

```

Figure 4.6 : Example classes written in Python.

to Java as well. Since Java has explicit type declarations, the type-inference problem is much easier.

To describe the type-inference algorithm as it applies to user-defined types, we use the Python classes shown in Figure 4.6 as a running example (taken from [37]). `Shape` defines a method `move`. `Circle` is a subtype of `Shape` and therefore inherits the method `move`. `Circle` also defines a method `area`.

#### 4.8.1 Type Lattice

To perform type inference, a lattice of all types used by the libraries must be constructed. Figure 4.7 describes the type lattice corresponding to the example shown in Figure 4.6. The return type-jump-functions over this lattice for the methods `move` and `area` are shown in Figure 4.8.

While statement constraints for user-defined types are similar to the constraints for the primitive types in MATLAB, there is no bound on the size of the lattice. To combat the extra complexity caused by this, new ways to rewrite the statement

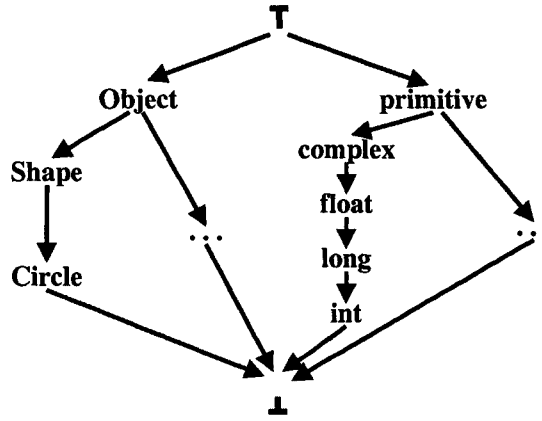


Figure 4.7 : Type Lattice for Python with user-defined types.

```

o=move(self,i1,i2)
(Circle ≤ τo ≤ Circle) ∧ (Circle ≤ τself ≤ Circle) ∧
    (⊥ ≤ τi1 ≤ real) ∧ (⊥ ≤ τi2 ≤ real) XOR
(Shape ≤ τo ≤ Shape) ∧ (Shape ≤ τself ≤ Shape) ∧
    (⊥ ≤ τi1 ≤ real) ∧ (⊥ ≤ τi2 ≤ real)

o=area(self)
(real ≤ τo ≤ real) ∧ (Circle ≤ τself ≤ Circle)

```

Figure 4.8 : Return type-jump-function for move and area.

constraints more concisely are considered.

For the user-defined types problem, the complexity of the algorithm,  $O(l^v n^2)$ , is unbounded because the number of elements in the lattice is unbounded. Since  $l^v$  represents an upper bound on the number of nodes at a particular level, steps can be taken to try to ensure that the number of nodes on a level never reaches these bounds. One such step involves using more sophisticated constraints that allow each clause (or node) to encode more information. For example, the return type-jump-function for move could be rewritten as shown in Figure 4.9. The types of the program variables are assigned type variables. These type variables are then constrained to

```

o=move(self,i1,i2)
( $\tau^o = \mathbf{T}$ )  $\wedge$  ( $\tau^{self} = \mathbf{T}$ )  $\wedge$  ( $\perp \leq \mathbf{T} \leq \mathbf{Shape}$ )
      ( $\perp \leq \tau^{i1} \leq \mathbf{real}$ )  $\wedge$  ( $\perp \leq \tau^{i2} \leq \mathbf{real}$ )

```

Figure 4.9 : Concise return type-jump-function for move.

ranges. Note that if the ranges in the old return type-jump-function were merged, the exact relationship of the variable types would be lost. The new return type-jump-function still captures all the information in the original (*i.e.*, the fact that the input and output type must be the same), but it requires only one node. In this way, all clauses that describe the same type pattern can be collapsed into a single node. Most methods with the same name would follow the same type pattern, but the algorithm is flexible enough to handle cases that do not have this property. The range that binds the type variables can also be written in terms of other type variables that define the types of one of the procedure variables. These type variables are similar to the  $\$$ -variables used in size inference.

With the new constraints, the number of nodes per level corresponds to the number of different patterns of the types for a particular procedure, not the number of possible type configurations. This significantly reduces the complexity of the algorithm. Although the number of viable type patterns can still be large, in practice only one type patterns should apply to a particular method name.

The intrinsic type and pattern problems can also benefit from this increased precision without an increase in the complexity.

#### 4.8.2 Extending the Lattice

An important feature of object-oriented programming is that the type hierarchy can easily be extended, and by doing so, the types over which methods work are also extended. This poses a problem for telescoping languages, since the type-inference algorithm assumes that the return type-jump-functions for the called procedure have



```

class Square(Shape):
    def __init__(self,x=0,y=0,side=1):
        Shape.__init__(self, x, y)
        self.side = side
    def area(self):
        return self.side * self.side

```

Figure 4.10 : Example type extension.

been formed before the calling procedure. Adding types after analysis has already been performed on a procedure invalidates the return type-jump-function. Note that for the purposes of telescoping languages, types are added only when new libraries that potentially use the existing libraries are added or when new types are defined in the user script.

There are two possible cases to consider. First, if the new types implement methods that have the same type pattern as already-defined methods, all that is required is an extension to the ranges allowed for the type variables in some of the clauses. In this case, the nodes in the graph do not change. The compiler would need to examine the new edges to see if they allow for more cliques in the graph than were previously possible.

For example, suppose a `Square` class is added, as shown in Figure 4.10. Then the new return type-jump-function for `area` would extend the range of the type variables to include `Square`.

Second, if the new types cause the methods to have different type patterns from the previous definitions of the method, new clauses must be added to the constraint. New nodes will need to be added to the graph corresponding to the new clauses as well as new edges to the rest of the graph.

Obviously, performing type inference again for all methods that are defined in or inherited by a new class as well as all methods that involve the new types indirectly would produce correct results. However, this is more expensive than necessary. One

optimization is to store the graph and cliques along with the generated variants. Then the new cliques would only have to be computed by checking edges from nodes involving the new types. While this could potentially double the storage requirement, it would make updating these constructs to include the new types more efficient.

Lattices can also be extended in the intrinsic-type and pattern problems if a library writer or application developer determines that a particular type is important, and it is not yet represented in the lattices. For example, a new sparsity pattern such as banded could be added to the pattern lattice. In these cases, similar strategies can be applied.

Changes in methods for specialization when this occurs are discussed in the next chapter.

## Chapter 5

# Type-Based Specialization

Once the types for a library procedure have been determined, the specialization engine must use this information to optimize the procedure. Specialization occurs during the code generation phase for both library compilation and script compilation.

There are two important aspects of specialization - variant generation and variant selection. Variant generation is performed during library compilation and, for the type-based prototype compiler, refers to generating the specialized Fortran or C variants as well as determining which specialized variants to generate given the inferred type configurations. Variant selection occurs during both library compilation and script compilation. During code generation, the compiler must determine the most appropriate variant with which to replace the operation or procedure call at each point in the code. This chapter discusses issues related to both aspects of specialization. First, we discuss the relationship between variant selection and generation in terms of the simplest model for type-based specialization. We will then discuss how the need to control the number of variants affects each aspect.

### 5.1 A Simple Model for Specialization

The simplest model for specialization generates a separate variant specialized for the types given by each type configuration over the parameters. Separate paths are generated within the variant that correspond to two distinct type configurations with the same parameter types (*i.e.*, the code within the procedure can be cloned at control-flow points if the types of the variables depends on which paths are taken). The paths are predicated on the types of the non-input parameter u-vars, so that the

appropriate path will be taken at runtime.

When generating each specialized variant or path in the procedure, the compiler replaces each operator or procedure call with a call to a specialized variant based on the types given in the corresponding type configuration. The process is called variant selection.

Each variant is stored in a database and is indexed by the corresponding clause in the return type-jump-function for the procedure, which details the range of input and output types handled by the variant. Thus, when specializing a calling procedure or script, the compiler can replace the call to the procedure by looking in the database for the variant with the appropriate type configurations over the parameters.

Because the return type-jump-function lists every possible type configuration over the input, any valid calling procedure will pass actual types to the procedure that correspond to a single clause in the return type-jump-function. Since it is assumed in the simple model that there is a single variant corresponding to each possible type configuration over the inputs, the variant called is the one with the type signature that corresponds exactly to the types given for the calling procedure in the type jump-function. Note that while there may be several variants that will handle the same type configuration, in the simple model, there is exactly one variant that matches the parameter types.

## 5.2 Variant Generation

Many of the variants generated by the simple model will not be used in practice. Furthermore, it is likely that some of the variants do not provide much added benefit in terms of performance over variants for more general types. To avoid code explosion, it is important to limit the variants generated.

To handle this problem, the compiler can determine that two or more type configurations in the type jump-function for a library procedure do not need to have distinct variants. The type of each variable in the merged type configuration is the bottom-

most ancestor on the lattice of the types for the variable given in each of the distinct type configurations. Note that since the type jump-function lists all legal types, the merged type configuration should always correspond to a type configuration in the type jump-function if the types can be legally merged. Thus, the merging process is actually just eliminating type configurations that are deemed not important by the compiler if another type configuration can handle the types.

It is still important that all inferred type configurations are covered in the variants, where “covered” means that there is a variant that will produce correct and efficient code if a particular type configuration occurs. Thus, for the size problem, all variants should be generated (unless annotations state otherwise), since none of the type configurations can legally cover another one without changing the meaning of the code. If it is determined that no operators that are overloaded based on the size of the variables occur within the procedure, then this requirement can be relaxed.

### 5.2.1 Beneficial Variants

In many cases, the benefit of generating a separate variant is small, since distinguishing between certain input types may have little effect on the execution speed of the code. For example, the parameter may only be used once or not at all within the code or may be used in a way that the exact type does not matter for performance. The compiler identifies the important variants to generate based on three factors: correctness, space, and profitability.

Variants may need to be generated to preserve the correctness of the code. For example, separate variants are needed for inputs that can either be scalars or non-scalars if the parameter is used in a multiplication, since this is a different operation depending on the type of the input parameter. Also, the compiler needs to generate variants that cover every possible range given in the return type-jump-function.

The result of the type-inference algorithm discussed in the previous chapter lists only the possible type configurations, although it does limit the number of configu-

rations to only those that could legally occur (without type errors) in practice given the information available at library-compilation time. However, it does not provide any analysis about which variants are beneficial to generate. The compiler must determine how often parameters may be used within the code and how they are used. We developed a *static-estimation strategy* that works in conjunction with the type-inference algorithm to provide information on which variants are the most beneficial to generate. Currently the static-estimation strategy is not part of TeleGen.

**Static Estimation and Type Inference** The information of interest in determining which variants to generate involves information about how each parameter is used in the procedure. This includes both the number of uses, and how advantageous specializing for each use is in terms of possible performance benefit. For example, if a parameter is passed to a procedure that is called within the loop and there exists a specialized version of the called procedures that works with the parameter of a specific type, then it may be important to generate a separate variant for this case.

This information can be built up from the primitive operators along with the type information with the static-estimation strategy. To statically estimate the benefit, each clause in the return type-jump-functions is assigned a score. For primitive operations, this score can be obtained from experimental data. Each clause has a corresponding library procedure that it uses in place of the operator to specialize for the types. These library procedures are used to determine the scores. The average number of cycles it takes to perform the operation for a given set of input data given the types in a clause is a good metric for the score for that clause. For example, in the return type-jump-function for multiplication, the clause that corresponds to integer multiplication would have a much lower score than the one corresponding to complex multiplication.

Determining the scores on the primitive operators is complicated by the fact that type inference is performed separately for each type problem. However, the library

procedures representing the primitive operations are defined in terms of all the type problems. Therefore, each clause in a return type-jump-function for a given type problem may actually have multiple associated library procedures corresponding to variations in types for the other type problems.

In determining the scores for individual type problems, it is difficult to factor out the effects of the other type problems. One way to do this might be, for each possible clause in the return type-jump-function for a given problem, to take the average of all corresponding library procedures (defining the operation) that satisfy the constraints of the given clause (*i.e.*, the procedures that range over the types for the other type problems). Averaging these scores should work well in practice, since the types of the variables have “steady” effects on the performance of the code. For example, it is unlikely that a complex version of the code will run faster than an integer version regardless of the shapes of the arrays.

The scores for the return type-jump-functions of library procedures are built up from the primitive operators and other library procedures. This building-up of scores works in conjunction with the type-inference algorithm. When the return type-jump-function is translated into a statement constraint, the scores from the clauses in the return type-jump-function are attached to the corresponding clauses in the statement constraint. For statements that are in a loop, the score can be multiplied by a factor of ten to handle that fact that it is important to specialize the pieces of code that occur most frequently. These scores are then attached to the appropriate nodes in the constraint graph. This information can be combined over each possible type configuration by simply summing the scores for each node in the corresponding clique.

These scores on the type configurations are then used to determine which variants are the most beneficial to generate in terms of the greatest wins in performance. The type configurations with the greatest differences in scores from the type configuration with the most general types represent variants that may be beneficial to generate. Type configurations with similar scores can be merged if possible.

The type configuration scores are also used to determine scores on the clauses in the return type-jump-function for the library procedures. If two or more type configurations correspond to the same clause in the return type-jump-function, the scores on the type configurations are averaged to determine the score for the corresponding clause. Therefore, all the scores can be built up from the scores of the primitive operations and other library procedures.

Note that this same method can be used to determine whether to generate specialized paths for types determined inside the procedure at points corresponding to non-input-determined procedures or control-flow decisions. If there is not much benefit to be obtained by specializing for the different possible types separately, the compiler can just merge the types to generate a single version of the path.

The advantage of this approach is that the relative benefit to generating each variant can be determined with reasonable accuracy before having to generate the actual variant. Because this approach can be done in conjunction with type inference, the added complexity is very slight, making it much more efficient than other techniques, such as actually generating the variants and running them on sample input, which in many cases is not available.

### 5.2.2 Most Used Variants

Another important factor in determining which variants are most beneficial is how often the procedure is called with each calling context.

Because, in the telescoping-languages framework, code generation for library compilation occurs before any possible calling contexts are known, it is impossible for the library compiler to determine from the code which type configurations will be used most in practice without the benefit of extra information. However, because the library compiler can make use of annotations from the library developer, the library writer can aid the compiler by providing an annotation stating that a specific calling context is expected to occur frequently in practice. Alternatively, the library writer



can provide a sample set of user-level scripts.

Type configurations that match annotations indicating a frequent occurrence should have a variant generated, even if the benefit is only very slight. The library writer can include an annotation on the frequency of a particular type configuration by attaching a use score to each clause in the annotation. The use score indicates a factor by which each type configuration involving that clause should be multiplied. The multiplier can be a number from 0 to 1, with 0 meaning that this case is almost always the case that would occur, and 1 being a case that would rarely occur. Thus, the type configurations most likely to occur in practice will have a lower overall score, which triggers the compiler to generate separate variants for these cases, since it looks to generate the variants that give the fastest times.

This will not only ensure that a variant is generated for the most-used cases, but it will also allow the compiler to determine the most likely type configurations for calling procedures. The occurrence of a particular procedure call will, in one sense, be able to provide hints to the compiler of the types that are most likely to occur. For example, the `ones` operator almost always has integer scalar inputs. Otherwise, in the MATLAB interpreter, a warning is generated, although the program will still execute. Therefore, a score of zero on the clause representing scalar inputs (and a score of one on the other clauses), would provide a hint to the compiler for calling procedures, that it is important to generate variants that call `ones` with scalar inputs.

### 5.3 Variant Selection

Whereas in the simple model for specialization in which every inferred type configuration corresponds to a distinct specialized variant or specialized path within a variant, once variants are merged, the problem of dispatching to the appropriate variant becomes more complicated. Specifically, it is now possible for more than one variant to be equally appropriate in terms of the parameter types. It is the responsibility of the library generator to set up a mechanism for determining the best variant.

As a side note, even before merging the variants, the simple model is not always viable. Specifically, if the procedure called is a primitive operation in the base language, the compiler must choose an existing library with which to replace the calls. For example, the matrix operations used in `ArnoldiC` are replaced with calls to the BLAS library. Therefore, the compiler is restricted to specialized variants given in the library, which may not include one variant for every possible type configuration on the parameters. In this case, the compiler must determine the best version to call.

Once variants have been merged, each handles a broader range of types. The problem occurs when there are two or more variants that are specialized for the types of different parameters. For example, assume there exists a library procedure with the header:

```
function [c] = lib_proc(a,b)
```

Assume that `lib_proc` accepts both real and complex inputs. Also assume that there are two variants for this procedure: `lib_proc1` that handles parameter `a` as real and `b` as complex and `lib_proc2` that handles parameter `a` as complex and `b` as real. If there is a procedure that calls `lib_proc` and passes it two reals, the question becomes which variant is best for that call site.

This question is answered by how `a` and `b` are used within `lib_proc`. Again, the compiler can use static estimation described in the previous section to determine which parameter is more important to specialize for (the version with the lowest score is used). Ties can be broken arbitrarily.

The cost of coercing the actual parameters to the types given in the type signatures of the generated variants can prove to be an important factor in determining which variant to choose. This is particularly true for array-valued actual parameters.

To maintain fast compilations of scripts, questions such as the one posed above will be answered during library generation time to the extent possible. The library generator has a list of all legal configurations of the input and output parameters in the form of the return type-jump-function. Therefore, the library generator can

determine which variant should be dispatched to for each possible type configuration and store this information in a table. Thus, to call the appropriate variant, the compiler only needs to look up the types given in the calling procedure or script in the table.

### 5.3.1 Selection from Output Types.

Because type inference handles both forward and backward flow of type information, variants can be selected based on the types of the output parameters for the procedure calls as well as the input types.

This means the compiler will find more opportunities to dispatch to variants statically than when using more traditional type-inference strategies. This is especially important for object-oriented types to enable transformations such as object inlining, which is itself an enabling optimization [14, 12].

To do this, variants need to be generated that assume specific output types. In case the output types cannot be determined by their uses in the calling program, variants should also be generated for the case when the exact output type may not be determined until runtime.

### 5.3.2 Coercion Problem

Because there may not be a single variant for each possible type configuration over the parameters, coercion might be necessary if none of the inputs accept the exact type passed in. Therefore the specialization engine may need to add coercion operations on the parameters of a procedure call. Coercion operations can be stored on the edges of the lattices for the type problems.

Coercion is also necessary when types are added to the lattice after type analysis and code generation for a called procedure has already been performed. This is especially likely in the pattern problem because new sparsity patterns with new types can be added.

These types must be coerced from the new type into an old type, specifically, the type corresponding to the bottom-most ancestor in the new lattice that is an element of the old lattice. This means that the lattices used to perform type inference must be stored (or at least the set of lattice elements). If there are two such ancestors, the score can be used to coerce the type into the type that will be most efficient based on the scores.

Note that there will always be a valid type to which the type can be coerced, since the variant generation process ensured that all valid type configurations were covered by the set of variants produced.

## Chapter 6

### TeleGen

We implemented both the type-inference strategy and type-based specialization in a prototype telescoping-languages compiler, TeleGen. Currently, TeleGen only includes analyses and optimizations that are based on types. However, we designed TeleGen to be easily extendable to multiple languages, analyses, and optimizations.

This chapter gives an overview of our implementation of type inference and type-based specialization in TeleGen. It then describes the experiments that we used to validate these strategies.

#### 6.1 Structure of TeleGen

TeleGen is implemented in C++. It is organized into two main pieces - a language-independent piece and a language-dependent piece. The language-dependent piece currently assumes that MATLAB is the source language. The language-independent piece includes type inference and type-based specialization and provides interfaces to the language-dependent piece. To use TeleGen with a different source language, the user merely needs to implement the interfaces.

TeleGen proceeds by first building the information about a subroutine into an AST and CFG. While TeleGen provides interfaces that do not require an AST or a CFG to be built to perform type inference, type-based specialization still relies on graphs. As the MATLAB AST is built, the operations are flattened or unfolded so that each statement contains only one operation. This is necessary for type inference as described in Chapter 3.

### 6.1.1 Type Inference in TeleGen

Interprocedural type inference is performed over an entire library. It invokes type inference for each subroutine so that the types for called procedures are inferred before type inference is performed over the calling procedures. Traversing the partial call graph of the library in reverse post-order increases the likelihood that return type-jump-functions are available for called procedures. The interprocedural type-inference routine also handles recursion if present in the library.

Type inference is performed on each individual routine in isolation. Only the annotation tables give information about the other routines in the library. First, statement constraints are built from the annotation table and the subroutine. During this process, subscript information is stored for later use in determining the relationship between program variables and sizes.

Once the statement constraints are formed, TeleGen solves for types using the algorithm for type inference in the presence of control flow discussed in Chapter 3 and Chapter 4. After types have been inferred over a block, the solver reduces the type to a statement constraint, which is appended to all successor blocks. If any variables defined in the successor block are constrained by the summary constraint, the piece of the summary constraint concerning that definition is removed to avoid falsely over-constraining the variables. For example, a variable may be defined on the first iteration to be scalar, but can grow into an array on subsequent iterations. However, the summary constraint along the back edge of the loop would constrain the variable to be a scalar.

### 6.1.2 Type-Based Specialization

Once a fixed point is reached and all the types are solved, the AST is replicated so that there is a separate graph for every type configuration inferred. It is at this point that arrays are merged back together and SSA is deconstructed.

Each AST is specialized according to the corresponding type configuration. Spe-

cialization occurs in the language-independent part of TeleGen. For each operation in the subroutine, the specializer looks up the appropriate version with which to replace the call from the types of the arguments using the specialization table. The AST is modified to store the new calls. The specialization annotations for the generated Fortran or C variants are then stored in the specialization table.

### 6.1.3 Code Generation

After specialization, code generation produces the Fortran or C code. The code generator is in the language-dependent piece of TeleGen. It interfaces with the ASTs and specializer so that other back-end languages may be easily attached to TeleGen. The code generator has two responsibilities. The first is to ensure that all calls to specialized routines meet the Fortran or C standard.<sup>1</sup> The second is to ensure that all the size variables and allocation statements are generated.

Allocations occur as early as possible. If the size of a variable depends on a procedure variable that is not defined until inside the loop, code is generated to test the size of the variable to see if reallocation is necessary. Because we are able to take loop bounds into account when performing size inference, we are able to allocate most variables a single time, usually outside of loops. Techniques such as slice hoisting could eliminate the need to reallocate almost entirely [18].

### 6.1.4 Limitations to Current Implementation of the Compiler

TeleGen is able to handle a majority of the MATLAB language. However, there are a few properties of MATLAB that TeleGen is not able to correctly evaluate at this time. We felt that none of these detract from the validation of the core techniques described in the dissertation. None of the changes we made to the MATLAB benchmarks changed

---

<sup>1</sup>Currently, the C generator is only partially working and the Fortran generator produces Fortran 90.

the meaning of the code (although results do vary due to rounding error). We plan to handle the whole MATLAB language in the future.

First, MATLAB does not require that variables be defined on all paths that are reachable by their uses. TeleGen, however, does require this to build SSA. Therefore, for the benchmarks we used to evaluate TeleGen, where necessary, we added assignments to the variables (initialized to zero) at the beginning of the routines. This is an exception to our claim that we do not change the meaning of the code. If the subroutine is called and a path is taken that results in an output not being defined, in the original benchmark, MATLAB would result in an error, whereas in the generated code, zero(s) would be returned.

TeleGen currently does not allow multiple operations in the condition statements of loops or if statements. The flattening function is not able to push statements across basic block boundaries. This should be relatively easy to fix.

MATLAB allows users to assign ranges to variables. Ranges can then be passed in to loop headers or as subscript ranges for performing operations on sections of the arrays. Since type inference examines each operation in isolation, range variables are not understood by type inference. We inlined the ranges into the arrays and loops.

The current type-inference implementation includes a simple solver to solve  $\$$ -variables in each clique for size inference. This solver assumes only simple equations. Therefore, we are currently unable to handle concatenation, since concatenated variables have sizes that are the sum of the sizes of the input arrays. Since concatenation may occur in loops, this requires the solver to handle coefficients. While we do have a strategy to handle such equations, which we described in Chapter 3, we have not yet implemented it. We have replaced concatenation with assignment to array sections.

It appears as though MATLAB does not use the ATLAS-tuned BLAS for matrix multiplication. We found that there was a slight discrepancy in the result of matrix multiplication between MATLAB and the BLAS routines. In some of the benchmarks involving long chains of multiplication over many loop iterations, this rounding error



compounded to more significant differences in the results. Were we to have access to the MATLAB libraries, we would not have this problem. We did not feel that a difference in library implementation detracted from the results.

Currently, TeleGen uses the simple strategy to determine when to generate variants (*i.e.*, a single variant for every type configuration). We leave the implementation of the more sophisticated variant-generation techniques as future work.

## 6.2 Experimental Evaluation

We used TeleGen to evaluate the type-inference and type-based specialization strategies over library routines. Currently, TeleGen does not handle script compilation since it has no way to statically determine the exact types of matrices defined externally. TeleGen does not yet have the mechanism to generate runtime queries to disambiguate the types of these matrices to determine which specialized variant to call. The library procedures, however, are generated completely automatically, since types are determined by the operators, and multiple variants are generated to handle all cases. The script compiler should only generate a single variant. Therefore, to run the experiments, we hand wrote the Fortran scripts.

### 6.2.1 Experimental Setup

The benchmarks we used to evaluate TeleGen are MATLAB routines that were used in the development of the ARPACK library. These routines implement the core algorithms used in ARPACK.

All experiments were conducted on a two processor, 2.5 GHz PowerPC with 8 GB of memory. The L2 cache for each processor is 512 KB. The Fortran 90 code was compiled with the IBM XLF compiler and `-O3` optimization option. We used MATLAB 7 to run the MATLAB benchmarks.

To measure performance for all the Fortran and MATLAB experiments, we used wall-clock time, with granularity of a hundred-thousandth of a second. The times are

shown in seconds and are measured over multiple input matrices of varying size. We ran each experiment 5 times for benchmarks with running times around two hours or more (excepting the MATLAB LUFAC benchmark with cry10000, which we ran twice) and 10 times for the rest of the benchmarks and took the average of the non-outliers. We found at most one outlier for ten runs and none for five. The matrices are all represented as dense, real, and non-symmetric.

TeleGen is compiled with gcc 3.3 on the PowerPC with -O3 optimization. Timings of type inference in TeleGen are taken using the wall clock time as well and are averaged over five runs.

TeleGen replaces calls to the MATLAB primitive operators with calls to high-performance libraries. The primitive operators in the ARPACK development code were primarily matrix operations that corresponded to procedures in the BLAS library. We use ATLAS-tuned BLAS so that the code could be specialized for the architecture as well as the types [81]. ARPACK also uses ATLAS-tuned BLAS to perform most of the underlying matrix operations.

Experiments were run using matrices of varying sizes from MatrixMarket [61]-Platzmans Oceanographic Model (a 362 x 362 matrix with 5,786 non-zero entries), BCSSTK27 (a BCS Structural Engineering Matrix, which is 1224x1224 with 28,675 non-zero entries), BCSSTK28 (another BCS Structural Engineering Matrix, which is 4410x4410 with 111,717 non-zero entries), and CRY10000 (a Diffusion Model Study for Crystal Growth Simulation, which is non-symmetric, 10000x10000 with 49,699 non-zero entries). For inputs to the complex variants, we used QC324 (a Model of H<sub>2</sub><sup>+</sup> in an Electromagnetic Field, which is 324 x 324 with 26730 non-zero entries) and DWG961A (a Dispersive Waveguide Structure, which is 961x961 with 3405 non-zero entries). We also generated matrices for the QR factorization codes from drivers provided in the MATLAB development code suite.

To perform experiments for dense matrices, the matrices above were expanded into dense format. Similarly, to experiment with complex matrices, the representation

of the elements of the matrix were changed. In this way, the value of accurately determining the types can be seen.

## 6.2.2 Evaluation of Type-Inference Algorithm

	<i>ArnoldiC</i>	<i>Lanczos</i>	<i>LUfacC</i>	<i>QRcgsF</i>	<i>QRgivens</i>	<i>Totals</i>	<i>QRgivens</i>	<i>Givens</i>
Num Input Args	3	3	1	1			2	2
Num Phi-Nodes	4	5	5	7			13	3
Num Operations	29	26	27	44			92	23
Type Inference Time (seconds)	4.02	3.12	6.12	170.94		970.98	966.26	0.94
Total Num Configurations	60	15	6	8			28	36

Figure 6.1 : Results of type inference.

	<i>ArnoldiC</i>	<i>Lanczos</i>	<i>LUfacC</i>	<i>QRcgsF</i>	<i>QRgivens</i>	<i>Totals</i>	<i>QRgivens</i>	<i>Givens</i>
Num Input Args	3	3	1	1			2	2
Num Phi-Nodes	4	5	5	7			13	3
Num Operations	29	26	27	44			92	23
Size Inference Time (seconds)	1.80	1.62	3.09	158.67		378.65	378.47	0.18
Size Num Iterations	3	3	5	14			8	2
Num Size Configurations	3	3	2	4			4	4

Figure 6.2 : Results of size inference.

Figure 6.1 shows the results of type inference over the MATLAB benchmarks. The figure shows that type inference infers a small number of type configurations for the benchmarks in a reasonable amount of time given the complexity of the problem. Figures 6.2, 6.3, and 6.4 split out the type inference results for the size, intrinsic-type, and shape problems respectively.

The number of type configurations inferred for each type problem is well under the upper bound of  $l^u$  or the number of types in the type lattice to the number of

	<i>ArnoldiC</i>	<i>Lanczos</i>	<i>LUfacC</i>	<i>QRcgsF</i>	<i>QRgivens</i>	<i>Totals</i>	<i>QRgivens</i>	<i>Givens</i>
Num Input Args	3	3	1	1			2	2
Num Phi-Nodes	4	5	5	7			13	3
Num Operations	29	26	27	44			92	23
Intrinsic Inference Time (seconds)	1.56	1.24	2.74	10.83	586.94	586.25	0.69	
Intr Num Iterations	3	3	4	4			7	2
Num Intrinsic Configurations	5	5	3	2			7	9

Figure 6.3 : Results of intrinsic-type inference.

	<i>ArnoldiC</i>	<i>Lanczos</i>	<i>LUfacC</i>	<i>QRcgsF</i>	<i>QRgivens</i>	<i>Totals</i>	<i>QRgivens</i>	<i>Givens</i>
Num Input Args	3	3	1	1			2	2
Num Phi-Nodes	4	5	5	7			13	3
Num Operations	29	26	27	44			92	23
Shape Inference Time (seconds)	0.51	0.10	0.18	0.37	1.60	1.54	0.06	
Shape Num Iterations	3	3	4	3			4	2
Num Shape Configurations	4	1	1	1			1	1

Figure 6.4 : Results of shape inference.

$u - vars$  (the number of input variables and phi-nodes). Note that the number of lattice elements that TeleGen uses for the size, intrinsic type, and shape problems are two, seven, and six respectively. For the intrinsic type problem, since the benchmarks use only numerical operations, the number of lattice elements considered is actually only three (integer, real, and complex).

**ArnoldiC** The results of size inference for ArnoldiC are given in figure 3.2 and discussed in Chapter 3.

For intrinsic-type inference for ArnoldiC, there are three input parameters and three possible numerical types (integer, real, and complex). Therefore, barring  $u - vars$ , there is a maximum of nine possible intrinsic-type configurations. However, the algorithm is able to infer that the second input is always integer. The third

input parameter,  $v$ , is immediately killed with the statement,  $v = v/norm(v)$ ; Since  $v$  is used in a division, type inference does not distinguish whether the inputs to division are real or integer. Therefore, type inference infers that the input  $v$  must be either complex or real/integer. The first parameter,  $A$ , is used in the statement  $w = A * v$ ; Type inference does not distinguish whether an input to multiply is real or integer if the other input to multiply is complex. Therefore, if  $v$  is complex,  $A$  can either be complex or real/integer, which accounts for two of the five inferred type configurations. The other three configurations assign  $v$  to real/integer and  $A$  can be any of the three numerical types. Note that type inference considers real and integer together for multiplication and division because of the return type-jump-functions. Since these operations are intrinsic to MATLAB, to make separate cases for real and integer, we just need to redefine the return type-jump-functions. We did not see any cases where this would be necessary for performance.

Shape inference is a special case, since, currently, TeleGen relies on the library writer to provide annotations. We only annotated ArnoldiC with shape information. Otherwise, we did not use any annotations that constrained the types for any of the benchmarks. These shape annotations on  $A$  are propagated down to  $w = A * v$ , so that the appropriate multiplication variant for the shape is called.

**Lanczos** Lanczos is very similar in structure to ArnoldiC. Therefore, the inferred types, as well as the time for type inference are similar.

**LUfacC** LUfacC takes in only one input. Therefore, it makes sense that size inference produced two type configurations (one in which the input is scalar and one in which the input is non-scalar) and three intrinsic-type configurations corresponding to the three numerical type possibilities for the input.

**QRcgsF** QRcgsF also only has a single input. Therefore, the four inferred size configurations for QRcgsF exceed the maximum number of size configurations based

on the input. In fact, in all four size configurations, the input must be non-scalar. The difference in the size configurations comes primarily from control-flow paths taken through the routine. Two of the type configurations are inferred over the same control-flow path. However, they diverge at the statement,  $r = Q' * A(:, j);$ . Notice that the return type-jump-function for  $*$  is actually not type input-dependent, since two non-scalars can produce either a non-scalar or a scalar (multiplication of two vectors). Therefore,  $r$  may either be scalar or non-scalar if  $A$  and  $Q$  are both non-scalar. Since there are no uses of  $r$  that constrain it further (unlike the other benchmarks such as ArnoldiC), the type configurations diverge at this point.

TeleGen makes these four type configurations separate variants. Two of the type configurations assign the input to a vector and two assign it to a two dimensional matrix. All four type configurations have different type assignments to the output. Therefore, calling procedures may be able to determine which specialized variant to call based on the uses of the output. A general case still needs to be produced to catch cases when the output is not used in a type-significant way in the calling procedure.

There are only two intrinsic type configurations inferred for QRcgsF stating that the input parameter can be either complex or integer/real. Again, this is due to the fact that the input parameter is used in division.

Notice that the time to infer types, particularly size goes up for QRcgsF. This is primarily due to the increased complexity of the code as is evidenced by the number of phi-nodes as well as the number of operations. The number of phi-nodes plus the number of inputs gives a rough estimate on the number of u-vars. More importantly, the number of phi-nodes determines, in part, the number of iterations over the CFG until a fixed-point is reached. This is the primary factor in the increased time for QRcgsF, although, since the algorithm is  $O(n^2)$  on the number of operations, the larger number of operations plays a factor as well.

**QRgivens** QRgivens calls Givens inside a loop. Givens is a very simple routine with no loops, which is why it converges in two iterations for all the type problems. The second iteration is merely to check that a fixed point has been reached. The four size configurations cover all possible combinations of scalar or non-scalar types over the two input variables. The nine intrinsic-type configurations similarly range over all possibilities of the three numerical types over the two input parameters.

Size inference is able to infer for QRgivens that the two input parameters are either both matrices or both vectors of the same size. There are seven intrinsic-type configurations instead of the nine maximum configurations over the two inputs and three numerical types. Again, this is because for matrix multiplication, integers and reals are not distinguished if the other parameter is complex. This matrix multiplication does not happen directly on the inputs, but the types are propagated back to the inputs.

Note that although the intrinsic-type problem has more lattice elements, for most of the benchmarks, it actually takes less time than the size-inference problem. This is in part due to the more complicated solver required for the size inference problem, but also has to do with the number of iterations required until a fixed point is reached. The intrinsic-type constraints can be merged so that multiple types can be represented in a single constraint. Therefore, because there are fewer constraints at the join points, intrinsic-type inference converges in fewer iterations than size inference, where the equations cannot be merged together if a variable is an array on one edge and a scalar on the other. The one exception to this is in QRgivens. Even though the intrinsic-type problem converges in fewer iterations than size inference, the overall time spent inferring intrinsic types is much greater than the time inferring sizes. This is due, at least in part, to the fact that most of the variables are subscripted array accesses, which, as we discussed in Chapter 4, can greatly reduce the complexity of size inference by reducing the number of clauses per statement constraint. Intrinsic-

type inference does not benefit from subscript information.

The total number of configurations represents the number of variants TeleGen generates for each benchmark. For these benchmarks, the number of variants is relatively small. If the number of variants gets out of hand, then the strategies proposed in Chapter 5 can be employed.

<i>Benchmark</i>	<i>Matrix</i>	<i>Generated</i>	<i>MATLAB</i>	<i>Ratio</i>
ArnoldiC	plat362 (362x362)	0.02	0.05	2.86
	bcsstk27 (1224x1224)	0.15	0.24	1.65
	bcsstk28 (4410x4410)	2.19	2.54	1.16
	cry10000 (10000x10000)	11.06	12.04	1.09
	qc324 (324x324 complex)	0.03	0.11	4.36
	dwg961 (961x961 complex)	0.23	0.51	2.28
Lanczos	plat362 (362x362)	0.02	0.03	1.88
	bcsstk27 (1224x1224)	0.14	0.17	1.20
	bcsstk28 (4410x4410)	2.20	2.24	1.02
	qc324 (324x324 complex)	0.02	0.06	2.80
	dwg961 (961x961 complex)	0.22	0.43	1.91
LUfacC	plat362 (362x362)	0.11	1.36	12.10
	bcsstk27 (1224x1224)	4.90	30.18	6.16
	bcsstk28 (4410x4410)	230.03	4102.43	17.83
	cry10000 (10000x10000)	2748.51	> 50000	> 18.19
	qc324 (324x324 complex)	0.16	1.49	9.36
	dwg961 (961x961 complex)	4.55	25.44	5.59
QRcgsF	(200x160)	0.02	0.10	6.15
	(400x320)	0.33	1.06	3.24
	(1600x1280)	28.63	54.18	1.89
	(3200x2560)	223.68	429.56	1.92
	(6400x5120)	1708.55	3395.07	1.99
	QRgivens	(200x160)	0.04	0.93
	(400x320)	0.69	5.10	7.39
	(1600x1280)	361.41	669.10	1.85
	(3200x2560)	7008.92	12082.85	1.72

Figure 6.5 : Comparison of MATLAB performance versus automatically generated Fortran.



### 6.2.3 Code Generation

Figure 6.5 compares the running times of the MATLAB benchmarks to those of the automatically-generated Fortran. The automatically-generated Fortran routines are correct with respect to the MATLAB code.

While it is not the purpose of this thesis to prove that TeleGen always produces faster code than MATLAB, it is interesting to note that there can be a large performance benefit in moving from a high-level interpreted language to a lower-level explicitly-typed language where arrays are allocated as early as possible (most variables are only allocated once).

ArnoldiC includes the statement  $h = V(:, 1 : j)' * w$ ; inside the loop, where  $j$  is an induction variable in the loop. While many MATLAB programmers have adopted the practice of pre-allocating arrays by assigning the array an array of zeros of the maximum size seen in the procedure <sup>2</sup>,  $h$  cannot be preallocated without changing the meaning of the statement (or requiring subscripts). Therefore  $h$  is one example of an array that must be reallocated on every iteration of the loop. This means that although MATLAB is calling efficient library routines underneath, the cost of reallocation is not dominated by the time spent in the library routines.

The ratio between MATLAB and Fortran decreases slowly as the size of the input matrices grow for ArnoldiC and Lanczos, since the time spent in library routines on bigger matrices begins to dominate the interpretive overhead. For LUfacC, however, this is not the case due to the fact that LUfacC does not rely on the BLAS or equivalent libraries the way ArnoldiC and Lanczos do. Therefore, the interpretive overhead is more apparent.

In QRcgsF and QRgivens, the benefit of Fortan decreases more dramatically as the sizes of the matrices increase. This is again due to the reliance of these two benchmarks on BLAS-equivalent library routines.

---

<sup>2</sup>zeros not only pre-allocates the array, but ensures contiguous memory for the array

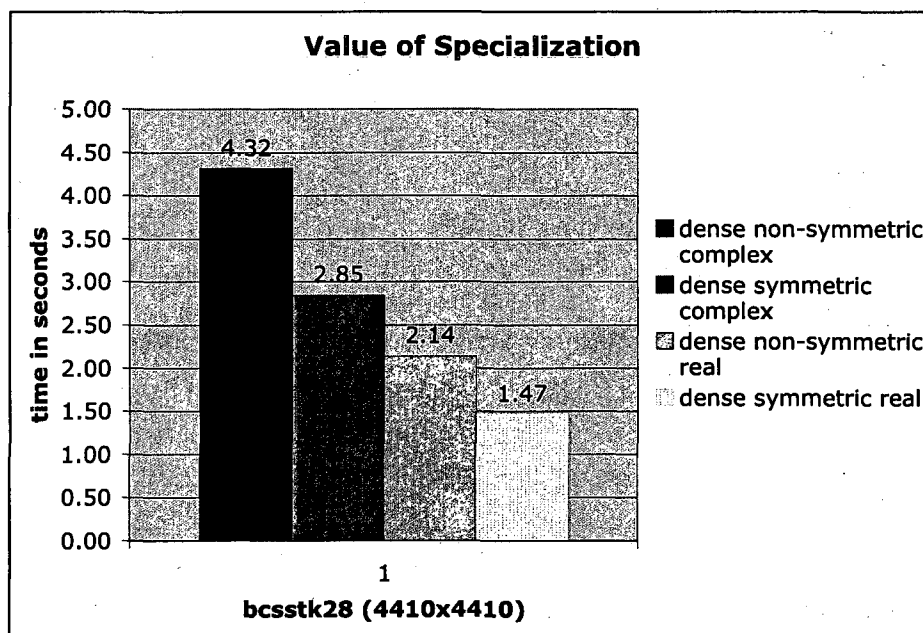


Figure 6.6 : Value of Specialization.

#### 6.2.4 Value of Specialization

Figure 6.6 demonstrates that type-based specialization can achieve large performance benefits for numerical applications. To perform type-based specialization, TeleGen needs a powerful static type-inference strategy that can allow the compiler to take advantage of all possible runtime types. Thus, this section not only validates the type-based specialization strategies described in this chapter, but also motivates the type-inference algorithm.

Figure 6.6 shows running times of different automatically specialized variants from TeleGen for the same input matrix. This experiment was designed to demonstrate

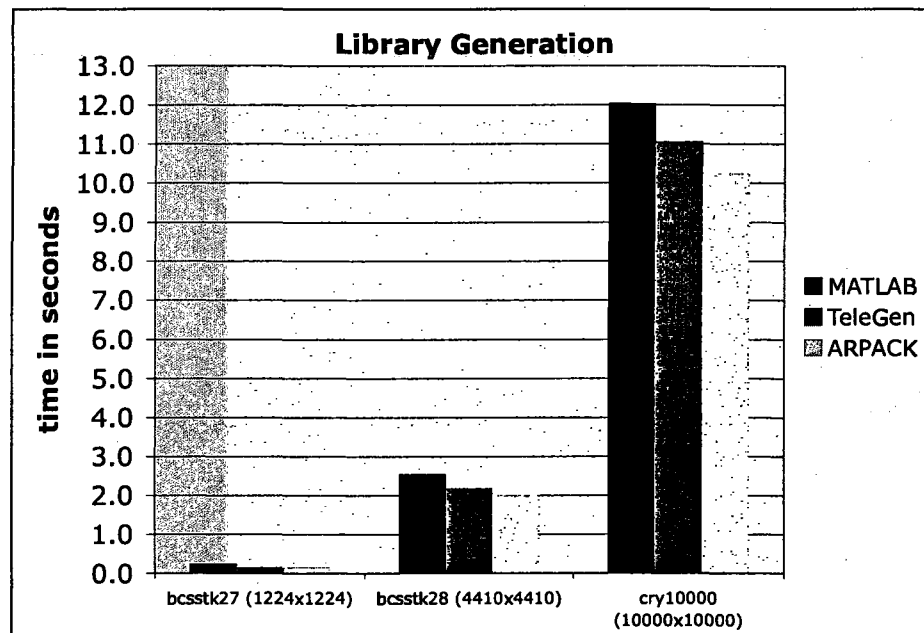


Figure 6.7 : Comparison to hand-coded Fortran.

that type-based-specialization is very important to achieve the maximum performance. An imprecisely inferred type can potentially result in a large performance loss.

### 6.2.5 Library Generation

ARPACK is a linear-algebra library designed to solve large-scale eigenvalue problems.

Figure 6.7 compares the execution time of ArnoldiC under MATLAB 7 with the TeleGen and ARPACK versions for different matrix input sizes. The performance of the code from TeleGen is comparable to that of the hand-coded Fortran, and these

results scale as the size of the input matrix grows. Note that this is only true because ARPACK relies heavily on the BLAS routines, so determining which BLAS routine to call (*i.e.*, disambiguating types) as well as preallocating the arrays are the two most important optimizations in the hand-coded ARPACK. We are not claiming to have solved the library generation problem in TeleGen for libraries in general, but merely want to show that type inference and type-based specialization play an important role in telescoping languages, and that our type-inference strategy is sufficient to handle the typing needs of library generation in telescoping languages.

## Chapter 7

### Contributions and Future Work

As architectures become increasingly complex, the problem of encoding scientific applications that take full advantage of the platforms available becomes even more difficult, especially given the small number of expert programmers available. Alleviating the difficulty in code development for scientists is important to accelerate scientific advancement. Telescoping languages aims to allow scientists to easily build, maintain, and use languages that inherently understand the domain in which the scientist is working. This dissertation provides essential technology necessary for a telescoping-languages system, namely, a type-inference system that works within the unique constraints imposed by the telescoping-languages strategy and type-based specialization that optimizes based on the inferred types.

The telescoping-languages strategy envisions generating specialized variants of library routines based on properties of the variables for likely calling contexts. Therefore, type inference is essential to telescoping languages since the definition of types used for the purposes of this dissertation is categories of program variable properties. This can include anything from intrinsic types to ranges of values for the variables. To describe the type-inference system, we focused on variable size, intrinsic type, and array-shape type problems, since we used MATLAB as our example language to encode the libraries. However, the type-inference strategy can be applied to many more type problems than these, including value ranges, which will be important for further optimization.

## 7.1 Contributions

Telescoping languages envisions generating the optimizing compilers before all calling contexts for the libraries are necessarily known. This means that type inference must infer the most general types possible. However, as we showed, inferring the most general types will lead to poor performance of the generated code. Therefore, we need a type-inference system that will guide the library generator in the telescoping-languages compiler to produce multiple variants for different calling contexts. To represent this information, we defined the notion of *return type-jump-functions* and *type jump-functions*, which provide a mechanism for both describing the type information in detail and determining which type configurations should induce their own variant. Inferring types in TeleGen thus becomes the process of determining return type-jump-functions and type jump-functions for the library procedures.

This dissertation develops the algorithm for performing type inference in a telescoping-languages system. The type-inference algorithm is able to determine all possible type configurations and exact relationships between the types of the program variables to produce type jump-functions and return type-jump-functions. Through representing the algorithm using graphical constructs, we were able to prove that type inference is performed in  $O(n^2)$  time under certain reasonable conditions. We implemented the type-inference algorithm in TeleGen and demonstrated that the actual running time of the type inference is reasonable for examples of MATLAB development code. We also showed that type inference was able to infer the correct type configurations for these benchmarks and was precise. In other words, the number of type configurations was small enough to limit the number of variants that need to be generated.

We described how to extend the type-inference algorithm to handle multiple type problems that may be useful for other languages and even MATLAB. While some of these extensions are straightforward, some will require more work to exactly determine how to map the problem into the type-inference system. However, the type-inference

system is very flexible, which is important for building future telescoping-languages technology.

Once types are available, the library generator (or script compiler) in a telescoping-languages compiler needs to be able to utilize this information to specialize the variants. Type-based specialization involves two interrelated components - variant generation and variant selection. We generate variants by examining the original library routines and performing variant selection to determine the best variant for the types of the arguments at each call site in the routine. Once variant selection has completed, the code for the variant is generated.

Finally, we incorporated the type inference and type-based specialization into the prototype telescoping-languages system, TeleGen. In addition to providing a compiler for MATLAB scripts, TeleGen also performs library generation. TeleGen provided a framework for validating the type-inference and type-based specialization strategies. Because we implemented type inference and type-based specialization to be independent of the scripting language, we expect that TeleGen will provide a backbone for future type-based optimizations and future telescoping-languages systems.

## 7.2 Future Work

TeleGen provides only a basic type-based specialization engine. There are many optimizations and analyses that could provide opportunities for improvement in the library generator. Type inference itself can be extended to enable many of these optimizations and is applicable to many interesting problems. Additionally, the telescoping-languages strategy has several interesting applications.

**Avoiding Impact of Temporaries** Because both type inference and specialization require a single operator per statement, temporaries are an unavoidable side effect of the generated code. Temporaries introduce additional overhead as they have to be allocated, reallocated, and in many cases copied back into the program vari-

ables. Baumgartner et. al. have worked on mitigating the effects of intermediate arrays in tensor-contraction computations through what amounts to scheduling and tiling to avoid producing and consuming the entire array [10]. Added benefits of this strategy include reduction in the memory footprint and improved cache reuse as well as improvements to the instruction pipeline.

**Object-Oriented Inference** The ability to precisely infer user-defined types in object-oriented languages can lead to many performance improvements. In this dissertation, we sketched out some extensions that will be necessary to efficiently infer types for object-oriented languages in a telescoping system. We would like to implement these extensions to study their effectiveness and explore more potentially beneficial extensions to the algorithm as it is applied to user-defined types. Precise type inference can enable transformations such as object inlining [53], which is itself an enabling optimization.

**In-Placeness Algorithms** We have worked with researchers from National Instruments to come up with strategies for avoiding unnecessary copies in LabView [1]. Our heuristics use copy-avoidance advantage scores on potential copy pairs (*i.e.*, input and output combinations) to determine the most beneficial pairs to compute in place. We plan to use the size-inference strategy to better approximate the copy avoidance advantages. Currently, our research in LabView has focused on single-procedure optimization. We plan to use a telescoping-style strategy to summarize advantage information across procedures.

**Component Integration** Telescoping languages can be thought of as a component integration system in that the user scripts assemble components in the form of libraries to generate applications [1]. Also, the scripting language is independent of the languages used to write the libraries. We would like to research ways in which TeleGen can be extended into a component-integration system. We plan to use Fortress, one



of the high-productivity computing systems languages developed at Sun, which has language support for component integration, as a basis for exploring research issues in this area. We have been working with researchers for Sun to explore other ways in which Fortress could benefit from the type-based technologies found in TeleGen.

**Type Feedback** The type-inference algorithm presented in this dissertation can be used to provide feedback to library writers. Since it infers all valid type configurations, library writers can examine the return type-jump-function to ensure that their code handles the types that they expected. This will allow library writers to in turn feed information back to the compiler to more precisely infer types.

**Type Inference in Programming Languages** We would like to further explore the relationship between our type-inference algorithm and type inference-strategies from the programming-languages community. We feel that our algorithm solves some similar problems to those being studied in that community. To make a direct comparison to other type-inference strategies, we will need to extend CORE-MATLAB with more interesting features found in other programming languages.

## Appendix A

This appendix supplements the formal description of the type-inference algorithm described in Chapter 3. First, we describe the error-propagation rules for CORE-MATLAB. These rules help determine the validity of CORE-MATLAB programs. We prove safety with respect to the error propagation rules. We then formally define the normalization process to transform CORE-MATLAB programs so that each statement contains only one operation or procedure call and prove that this transformation preserves types. Finally, we prove the soundness and completeness of the type-inference solution with respect to the type system given in Chapter 3.

### A.1 Error Propagation Rules and Type Safety

**Theorem 3.5.1 Type Safety:**

Given Type, if  $\Gamma \vdash \Sigma, \Gamma \vdash e : t$  and  $\Sigma, e \hookrightarrow a$  then  $\Gamma \vdash a : t$ . Also, if  $\Gamma \vdash \Sigma, \Gamma \vdash P$ , and  $\Sigma, P \hookrightarrow \Sigma'$ , then  $\Gamma \vdash \Sigma'$ .

*Proof of Theorem 3.5.1:* By structural induction on evaluation derivation.

**Case [VAR]:** The result is immediate because  $x := a \in \Sigma$ , and  $\Gamma \vdash \Sigma$ . Therefore,  $\Gamma \vdash x : t$  means that  $\Gamma \vdash a : t$ , since  $x$  and  $a$  evaluate to the same value.

**Case [E-VAR]:** If  $x := a \notin \Sigma$ , then  $x$  is not well-typed.

**Case [ELEM]:** By the induction hypothesis  $\Sigma, e_1 \hookrightarrow a_1$ ,  $\Sigma, e_2 \hookrightarrow a_2$ , and  $\Sigma, x \hookrightarrow a$ , and  $\Gamma \vdash a_1 : t_1$ ,  $\Gamma \vdash a_2 : t_2$ , and  $\Gamma \vdash a : \langle s_1, s_2 \rangle$ . We know that the sizes of  $a_1$  and  $a_2$  are both  $\langle 1, 1 \rangle$ . Since we know that  $a_1 \leq s_1$  and  $a_2 \leq s_2$ , the evaluation results in  $a(a_1, a_2) : \langle 1, 1 \rangle$ , since we are only accessing a single element of the matrix.

$$\begin{array}{c}
\frac{\Sigma, e_1 \hookrightarrow \text{error} \text{ or } \Sigma, e_2 \hookrightarrow \text{error}}{\Sigma, x(e_1, e_2) \hookrightarrow \text{error}} \text{ [EP-ELEM]} \\
\\
\frac{\Sigma, e \hookrightarrow \text{error}}{\Sigma, x := e \hookrightarrow \text{error}} \text{ [EP-STMT]} \\
\\
\frac{\Sigma, c \hookrightarrow \text{error}}{\Sigma, c; b \hookrightarrow \text{error}} \text{ [EP-BODY]} \\
\\
\frac{\exists i \in \{1, \dots, n\} \text{ s.t. } \Sigma, e_i \hookrightarrow \text{error} \text{ or } \\
\text{Def}(f) = \text{fn } x_0 := f(x_1, \dots, x_n) \text{ b} \\
\Sigma, [x_1 \mapsto a_1, \dots, x_n \mapsto a_n] \text{ b} \hookrightarrow \text{error}}{\Sigma, f(e_1, \dots, e_n) \hookrightarrow \text{error}} \text{ [EP-APP]} \\
\\
\frac{\Sigma, e_1 \hookrightarrow \text{error} \text{ or } \Sigma, e_2 \hookrightarrow \text{error}}{\Sigma, e_1 + e_2 \hookrightarrow \text{error}} \text{ [EP-ADD]} \\
\\
\frac{\Sigma, e_1 \hookrightarrow \text{error} \text{ or } \Sigma, e_2 \hookrightarrow \text{error}}{\Sigma, e_1 * e_2 \hookrightarrow \text{error}} \text{ [EP-MULT]}
\end{array}$$

Figure A.1 : Error Propagation Rules

Case [E-ELEM]: By induction hypothesis  $\Sigma, e_1 \hookrightarrow a_1$ ,  $\Sigma, e_2 \hookrightarrow a_2$ , and  $\Sigma, x \hookrightarrow a$ , and  $\vdash a_1 : t_1$ ,  $\Gamma \vdash a_2 : t_2$ , and  $\Gamma \vdash a : \langle s_1, s_2 \rangle$ . If  $\{x := a\} \notin \Sigma$ , then this fails an assumption of the theorem and is therefore not well-typed. Also, if either  $t_1 \neq \langle 1, 1 \rangle$  or  $t_2 \neq \langle 1, 1 \rangle$ , then this expression is not well-typed.

Case [APP]: We know by the induction hypothesis  $\Sigma, e_i \hookrightarrow a_i$  that  $a_i : t_i \forall i \in 1, \dots, n$ . We also know that  $\sigma^{x_0} = t'_0 \wedge \dots \wedge \sigma^{x_n} = t'_n \in \text{Type}(f)$ . Since  $\text{Type}(f)$  depends on the function type for  $f$ , we know by the induction hypothesis that there is a valid evaluation of the body  $f$  with the given input types, where  $x_0$  has type  $t_0$ . Therefore, since  $a$  gets the value and therefore type of  $x_0$ ,  $a : t_0$ .

Case [E1-APP]: If the wrong number of arguments are given then no typing assignment to arguments will be in  $\text{Type}(f)$ . Therefore, this expression is not

well-typed.

Case [E2-APP]: If  $f \notin Def$ , then there is no entry for  $f$  in table *Type*, which means that the expression is not well-typed.

Case [ADD]: By the induction hypothesis  $\Sigma, e_1 \hookrightarrow a_1$ ,  $\Sigma, e_2 \hookrightarrow a_2$ , and  $a_1 : t_1$  and  $a_2 : t_2$ .  $t_1$  and  $t_2$  must be the same if this is well typed. Therefore,  $a : t$ , by the definition of *Add*.

Case [E-ADD]: By the induction hypothesis  $\Sigma, e_1 \hookrightarrow a_1$ ,  $\Sigma, e_2 \hookrightarrow a_2$ , and  $a_1 : t_1$  and  $a_2 : t_2$ . Since  $t_1$  and  $t_2$  are not the same, this expression is not well typed.

Case [MULT]: By the induction hypothesis  $\Sigma, e_1 \hookrightarrow a_1$ ,  $\Sigma, e_2 \hookrightarrow a_2$ , and  $a_1 : \langle s'_1, s''_1 \rangle$  and  $a_2 : \langle s'_2, s''_2 \rangle$ .  $s'_1$  and  $s'_2$  must be the same if this is well typed. Therefore,  $a : \langle s'_1, s''_2 \rangle$ , by the definition of *Mult*.

Case [E-MULT]: By the induction hypothesis  $\Sigma, e_1 \hookrightarrow a_1$ ,  $\Sigma, e_2 \hookrightarrow a_2$ , and  $a_1 : \langle s'_1, s''_1 \rangle$  and  $a_2 : \langle s'_2, s''_2 \rangle$ . Since  $s'_1$  and  $s'_2$  are not the same, this expression is not well-typed.

Case [STMT]: By the induction hypothesis  $\Sigma, e \hookrightarrow a$  we have  $\Gamma \vdash a : t$ . Since  $\Gamma \vdash x := e$  by assumption on the theorem,  $\Gamma(x) = t$  and  $\Gamma \vdash e : t$ . Therefore, since  $\Gamma \vdash \Sigma$ , then  $\Gamma \vdash \Sigma + \{x := e\}$ .

Case [STOP]: Trivially true.

Case [BODY]: By the induction hypothesis  $\Sigma, c \hookrightarrow \Sigma'$ , we have that  $\Gamma \vdash \Sigma'$ . Then, also by the induction hypothesis, if  $\Sigma', b \hookrightarrow \Sigma''$  then  $\Gamma \vdash \Sigma''$ .

Error propagation: The error propagation rules are given in Figure A.1. These rules state that if an evaluation in the antecedent results in an error, then the evaluation of the expression results in an error. Since we have by induction hypothesis that any evaluation that results in an error will not occur in a well-typed expression, the evaluations in the antecedents must evaluate expressions

$$\begin{array}{c}
\frac{}{nrm[x := a] \rightarrow x := a} \text{[VAL]} \quad \frac{}{nrm[x := x'] \rightarrow x := x'} \text{[VAR]} \\
\\
\frac{x_1, x_2 \text{ fresh in } b}{nrm[x_0 := x(e_1, e_2)] \rightarrow nrm[x_1 := e_1]; nrm[x_2 := e_2]; x_0 := x(x_1, x_2);} \text{[ELEM]} \\
\\
\frac{x_1, x_2 \text{ fresh in } b}{nrm[x_0 := e_1 + e_2] \rightarrow nrm[x_1 := e_1]; nrm[x_2 := e_2]; x_0 := x_1 + x_2;} \text{[ADD]} \\
\\
\frac{x_1, x_2 \text{ fresh in } b}{nrm[x_0 := e_1 * e_2] \rightarrow nrm[x_1 := e_1]; nrm[x_2 := e_2]; x_0 := x_1 * x_2;} \text{[MULT]} \\
\\
\frac{x_1, \dots, x_n \text{ fresh in } b}{nrm[x := f(e_1, \dots, e_n)] \rightarrow nrm[x_1 := e_1]; \dots; nrm[x_n := e_n]; x := f(x_1, \dots, x_n)} \text{[APP]} \\
\\
\frac{}{nrm[\text{stop}] \rightarrow \text{stop}} \text{[STOP]} \\
\\
\frac{}{nrm[c; b] \rightarrow nrm[c]; nrm[b]} \text{[BODY]}
\end{array}$$

Figure A.2 : Normalization Rules.

that are not well-typed. Therefore, the expression in the error propagation rule must not be well-typed.

□

## A.2 Normalization and Type Preservation

We show that normalization preserves types.

**Theorem A.2.1** *If  $\Gamma \vdash P$ , then there exists  $\Gamma' \supseteq \Gamma$  such that  $\Gamma' \vdash \text{norm}[P]$ . Furthermore,  $\Gamma' \vdash P$ .*

Proof of Theorem A.2.1 : We prove the first statement in the proof by structural induction over the normalization rules.

Case [VALUE]: trivial, since the result of the normalization is the same as in the original statement.

Case [VAR]: trivial, since the result of the normalization is the same as the original statement.

Case [ELEM]: Let  $\Gamma'$  be such that  $\Gamma' \supseteq \Gamma$ , and  $\Gamma' \vdash x_1 : t_1$  and  $\Gamma' \vdash x_2 : t_2$ . Then,  $\Gamma' \vdash x_1 := e_1; x_2 := e_2; x_0 := x(x_1, x_2)$ . Therefore, by the induction hypothesis  $\exists \Gamma''$  such that  $\Gamma'' \supseteq \Gamma'$  and  $\Gamma'' \vdash \text{norm}[x_1 := e_1], \Gamma'' \vdash \text{norm}[x_2 := e_2]$ . Therefore,  $\Gamma'' \vdash \text{norm}[x_0 := x(e_1, e_2)]$ .

Case [ADD]: Let  $\Gamma'$  be such that  $\Gamma' \supseteq \Gamma$ , and  $\Gamma' \vdash x_1 : t_1$  and  $\Gamma' \vdash x_2 : t_2$ . Then,  $\Gamma' \vdash x_1 := e_1; x_2 := e_2; x_0 := x_1 + x_2$ . Therefore, by the induction hypothesis  $\exists \Gamma''$  such that  $\Gamma'' \supseteq \Gamma'$  and  $\Gamma'' \vdash \text{norm}[x_1 := e_1], \Gamma'' \vdash \text{norm}[x_2 := e_2]$ . Therefore,  $\Gamma'' \vdash \text{norm}[x_0 := e_1 + e_2]$ .

Case [MULT]: Let  $\Gamma'$  be such that  $\Gamma' \supseteq \Gamma$ , and  $\Gamma' \vdash x_1 : t_1$  and  $\Gamma' \vdash x_2 : t_2$ . Then,  $\Gamma' \vdash x_1 := e_1; x_2 := e_2; x_0 := x_1 * x_2$ . Therefore, by the induction hypothesis  $\exists \Gamma''$  such that  $\Gamma'' \supseteq \Gamma'$  and  $\Gamma'' \vdash \text{norm}[x_1 := e_1], \Gamma'' \vdash \text{norm}[x_2 := e_2]$ . Therefore,  $\Gamma'' \vdash \text{norm}[x_0 := e_1 * e_2]$ .

Case [APP]: Let  $\Gamma'$  be such that  $\Gamma' \supseteq \Gamma$ , and  $\Gamma' \vdash x_i : t_i$  for all  $i \in \{1, \dots, n\}$ . Then,  $\Gamma' \vdash x_1 := e_1; \dots; x_n := e_n; x := f(x_1, \dots, x_n)$ . Therefore, by the induction hypothesis there exists a  $\Gamma''$  such that  $\Gamma'' \supseteq \Gamma'$  and  $\Gamma'' \vdash \text{norm}[x_i := e_i]$  for all  $i \in \{1, \dots, n\}$ . Therefore,  $\Gamma'' \vdash \text{norm}[x := f(e_1, \dots, e_n)]$ .

Case [STOP]: trivial, since the result of the normalization is the same as the original statement.

Case [BODY]: By induction hypothesis, there exists a  $\Gamma' \supseteq \Gamma$  and a  $\Gamma'' \supseteq \Gamma$  such that  $\Gamma' \vdash nrm[c]$  and  $\Gamma'' \vdash nrm[b]$ . Therefore  $\Gamma''' = \Gamma' \cup \Gamma''$  gives us  $\Gamma''' \vdash nrm[c; b]$ .

The second statement of the theorem is trivial since  $\Gamma' \supseteq \Gamma$  □

### A.3 Soundness

Theorem 3.5.2 Soundness: Given *Type*, *Def*, and *P*, if  $\mu_T, \Gamma \vdash pc$  and  $nrm[P] : pc$  then  $\Gamma \vdash P$ .

Proof of Theorem 3.5.2 (By Induction Over Procedure Bodies): Base Case: Let *P* be stop. Then the derivation with respect to any  $\Gamma$  is:

$$\overline{\Gamma \vdash \text{stop.}}$$

Induction step: Assume *P* is a normalized procedure,  $\Gamma$  is well-formed with respect to *P*, and  $\Gamma$  allows for a valid type derivation of *P*. Let  $\Gamma'$  be another type environment. We can treat type environments as though they are existentially quantified, so that we can meaningfully combine type environments by conjoining them. Note that this corresponds to performing a remapping of the sizes before conjoining the type environments so that no \$-variables are shared across  $\Gamma$  and  $\Gamma'$ . Let  $\Gamma'' = \Gamma \wedge \Gamma'$  be well-formed with respect to *c; P*. Then, since  $\Gamma$  is a subset of  $\Gamma''$ , there must be a valid derivation of *P* with respect to  $\Gamma''$ .

$$\overline{\dots}$$

$$\overline{\Gamma'' \vdash P}$$

We must show that this derivation can be extended to include *c*. Let *c* be  $x := f(e_1, \dots, e_n)$ . Since  $\Gamma''$  is well-formed, there must be a mapping,  $\mu'_T$  from \$-variables to linear expressions over the \$-variables such that for one of the clauses in the

statement constraint corresponding to  $c$ ,  $\mu'_T$  does not take matrices to scalars. Let  $\mu_D$  be the mapping from  $\$$ -vars to  $\$$ -vars that was used in generating the statement constraints. Let  $\mu_T = \mu'_T \circ \mu_D$ . Then there exists a mapping,  $\mu'_T$ , that satisfies the restrictions for  $\mu_T$  to be valid. Therefore, we can use  $\mu_T$  to prove that there is a type-derivation over  $c$  with respect to  $\Gamma$ , and we can append this derivation to that of  $P$  to get the type derivation for  $c; P$ .

$$\frac{\frac{\frac{\Gamma''(x) = t \quad \sigma^{x_0} = t'_0 \wedge \dots \wedge \sigma^{x_n} = t'_n \in Type(f)}{\Gamma'' \vdash x : t} \quad \Gamma'' \vdash e_1 : t_1 \dots \Gamma'' \vdash e_n : t_n}{\Gamma'' \vdash f(e_1, \dots, e_n) : t_0} \quad \dots}{\Gamma'' \vdash x := f(e_1, \dots, e_n)} \quad \Gamma'' \vdash P}{\Gamma'' \vdash x := f(e_1, \dots, e_n); P}$$

Statements of the form  $x := a$ ,  $x := x'$ , and  $x := x'(e_1, e_2)$  are trivial, since they are the same constraints used in  $\Gamma$  for these statements.

When the expressions passed to  $f$  involve subscripts, extra constraints stating that these expressions have to have a size of  $\langle 1, 1 \rangle$  are added. These constraints match those given in the type system.

By theorem A.2.1, we have that  $\Gamma \vdash P'$ , where  $P'$  is the pre-normalization form of  $P$  if  $\Gamma \vdash P$ .

□

## A.4 Completeness

**Theorem 3.5.3 Completeness:** Given *Type*, *Def*, and  $P$ , if  $\Gamma \not\vdash P$  then there does not exist a  $\mu_T$  such that  $\mu_T, \Gamma \vdash pc$ , where  $nrm[P] : pc$ .

**Proof of Theorem 3.5.3 (By Contradiction):** Assume there exists a valid type derivation for the normalized procedure  $P$  with respect to  $\Gamma$ , where  $\Gamma$  is not well-formed with respect to  $P$ . Then for some statement,  $c$ , the statement constraint, for every clause, for some atom, there does not exist a mapping,  $\mu'_T$ , that can take the



atom to  $\Gamma$ . Therefore, there does not exist a type assignment in *Type* for  $f$  called at statement  $c$ , since there can be no valid mapping,  $\mu_T$ . Therefore by theorem A.2.1, we have that  $\Gamma \not\vdash P$ , means that if  $\Gamma \not\vdash P'$  where  $P'$  is the pre-normalization form of  $P$ .

□

## Bibliography

- [1] Samah Abu-Mahmeed. In-place updating of variables in labVIEW. Master's thesis, Department of Computer Science, Rice University, Houston, Texas, 2008.
- [2] Ole Agesen. Constraint-based type inference and parametric polymorphism. In *First International Static Analysis Symposium*, pages 78–100, Namur, Belgium, September 1994.
- [3] Ole Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *ECOOP '95: Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 2–26, arhus, Denmark,, 1995. Springer-Verlag.
- [4] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1985.
- [5] George Almási and David Padua. MaJIC: Compiling MATLAB for speed and responsiveness. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 294–303, Berlin, Germany, 2002. ACM.
- [6] Gheorghe Almási. *MaJIC: A Matlab Just-in-time Compiler*. PhD thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, 2001.
- [7] Rob Armstrong, Dennis Gannon, Al Geist, Katarzyna Keahey, Scott Kohn, Lois McInnes, Steve Parker, and Brent Smolinski. Toward a common component architecture for high-performance scientific computing. In *HPDC '99: Proceedings of the 8th IEEE International Symposium on High Performance Distributed*

- Computing*, page 13, Redondo Beach, California, USA, 1999. IEEE Computer Society.
- [8] P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Haldar, P. Joisha, A. Jones, A. Kanhare, A. Nayak, S. Periyacheri, M. Walkden, and D. Zaretsky. A MATLAB compiler for distributed heterogeneous reconfigurable computing systems. In *FCCM '00: Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, page 39, Napa Valley, CA, USA, April 2000. IEEE Computer Society.
- [9] F. Basseti and D. Quinlan. C++ expression templates performance issues in scientific computing. In *IPPS '98: Proceedings of the 12th. International Parallel Processing Symposium on International Parallel Processing Symposium*, page 635, Orlando, FL, USA, 1998. IEEE Computer Society.
- [10] Gerald Baumgartner, Alexander Auer, David E. Bernholdt, Alina Bibireata, Venkatesh Choppella, Daniel Cociorva, Xiaoyang Gao, Robert J. Harrison, So Hirata, Sriram Krishnamoorthy, Sandhya Krishnan, Chi-Chung Lam, Qingda Lu, Marcel Nooijen, Russel M. Pitzer, J. Ramanujam, P. Sadayappan, and Alexander Sibiryakov. Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. In *Proceedings of the IEEE*, volume 93, pages 276–292, February 2005.
- [11] Bradley Broom, Rob Fowler, and Ken Kennedy. KelpIO: A telescope-ready domain-specific I/O library for irregular block-structured applications. In *Proceedings of the First IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 148–155, Brisbane, Australia, May 2001. IEEE Computer Society Press.
- [12] Z. Budimlic and K. Kennedy. Static interprocedural optimizations in Java. Technical Report CRPC-TR98746, Rice University, 1998.

- [13] Zoran Budimlić, Mackale Joyner, and Ken Kennedy. Improving compilation of Java scientific applications. *International Journal of High Performance Computing Applications*, 21(3):251–265, 2007.
- [14] Zoran Budimlic and Ken Kennedy. JaMake: A Java compiler environment. In *LSSC '01: Proceedings of the Third International Conference on Large-Scale Scientific Computing-Revised Papers*, pages 201–209, Sozopol, Bulgaria, 2001. Springer-Verlag.
- [15] David Callahan, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Interprocedural constant propagation. *SIGPLAN Not.*, 21(7):152–161, 1986.
- [16] Sébastien Carlier and J. B. Wells. Type inference with expansion variables and intersection types in system E and an exact correspondence with  $\beta$ -reduction. In *PPDP '04: Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 132–143, Verona, Italy, 2004.
- [17] Arun Chauhan and Ken Kennedy. Optimizing strategies for telescoping languages: procedure strength reduction and procedure vectorization. In *ICS '01: Proceedings of the 15th International Conference on Supercomputing*, pages 92–101, Sorrento, Italy, 2001.
- [18] Arun Chauhan and Ken Kennedy. Slice-hoisting for array-size inference in MATLAB. In *Languages and Compilers for Parallel Computing*, College Station, TX, USA, 2003.
- [19] Arun Chauhan, Cheryl McCosh, Ken Kennedy, and Richard Hanson. Automatic type-driven library generation for telescoping languages. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 51, Phoenix, AZ, USA, 2003. IEEE Computer Society.

- [20] Stephane Chauveau and Francois Bodin. Menhir: An environment for high performance matlab. In *LCR '98: Selected Papers from the 4th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 27–40, Pittsburgh, PA, USA, 1998. Springer-Verlag.
- [21] K. Cooper, M. W. Hall, and K. Kennedy. Procedure cloning. In *Proceedings of the 1992 IEEE International Conference on Computer Language*, Oakland, CA, USA, 1992.
- [22] Keith D. Cooper, Mary W. Hall, and Ken Kennedy. A methodology for procedure cloning. *Computer Languages*, 19(2):105–117, 1993.
- [23] Keith D. Cooper and Ken 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.
- [24] M. Coppo and M. Dezani-Ciancaglini. An extension of the basic functionality theory for the lambda-calculus. *Notre Dame Journal of Formal Logic*, 21(4), 1980.
- [25] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Principal type schemes and lambda-calculus semantics. *R. Hindley and J. Seldin, editors, To H. B. Curry. Essays on Combinatory Logic, Lambda-calculus and Formalism*, pages 535–560, 1980.
- [26] M. Coppo and P. Giannini. A complete type inference algorithm for simple intersection types. In *17th Colloq. Trees in Algebra and Programming*, volume 581 of *Lecture Notes in Computer Science*, pages 102–123. Springer-Verlag, 1992.
- [27] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

- [28] 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.
- [29] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212, Albuquerque, Mexico, 1982.
- [30] Rowan Davies and Frank Pfenning. Intersection types and computational effects. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 198–208, Montreal, Canada, 2000.
- [31] J. Dean, C. Chambers, and D. Grove. Identifying profitable specialization in object-oriented languages. In *Partial Evaluation and Semantics-Based Program Manipulation*, Orlando, FL, USA, 1994.
- [32] Jeffrey Dean, Craig Chambers, and David Grove. Selective specialization for object-oriented languages. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 93–102, La Jolla, California, United States, 1995. ACM Press.
- [33] Luiz Antônio DeRose. *Compiler Techniques for Matlab Programs*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1996.
- [34] John W. Eaton. *GNU Octave Manual*. Network Theory Limited, 2002.
- [35] Daniel Elphick, Michael Leuschel, and Simon Cox. Partial evaluation of MATLAB. In *Generative Programming and Component Engineering (GPCE'03)*, Lecture Notes in Computer Science, pages 344–363, Erfurt, Germany, 2003.

- [36] Dawson R. Engler. Interface compilation: Steps toward compiling program interfaces as languages. *Software Engineering*, 25(3):387–400, 1999.
- [37] Chris Fehily. *Visual Quickstart Guide Python*. Peachpit Press, Berkeley, CA, 2002.
- [38] Mary Fletcher. Matlab D: Compiling parallel matlab with user-defined data distributions. Master's thesis, Department of Computer Science, Rice University, Houston, Texas, 2008.
- [39] Mary Fletcher, Cheryl McCosh, Guohua Jin, and Ken Kennedy. Compiling parallel matlab for general distributions using telescoping languages. In *Proceedings of ICASSP*, Honolulu, Hawaii, 2007. IEEE.
- [40] Mary Fletcher, Cheryl McCosh, Ken Kennedy, and Guohua Jin. Strategy for compiling parallel Matlab for general distributions. Technical Report TR06-877, Rice University, Houston, TX, 2006.
- [41] Tim Freeman and Frank Pfenning. Refinement types for ML. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 268–277, Toronto, Ontario, Canada, 1991.
- [42] Matteo Frigo and Steven 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, Seattle, WA, May 1998.
- [43] Dan Grove and Linda 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, Albuquerque, NM, June 1993.

- [44] S. Z. Guyer and C. Lin. Broadway: A compiler for exploiting the domain-specific semantics of software libraries. In *In Proceedings of IEEE*, volume 93, 2, pages 342–357, 2005.
- [45] Samuel Z. Guyer and Calvin Lin. An annotation language for optimizing software libraries. In *DSL'99: Proceedings of the 2nd conference on Conference on Domain-Specific Languages*, pages 4–4, Austin, Texas, 1999. USENIX Association.
- [46] Samuel Zev Guyer. *Incorporating domain-specific information into the compilation process*. PhD thesis, Department of Computer Science, The University of Texas at Austin, 2003. Supervisor-Calvin Lin.
- [47] J. Roger Hindley. Types with intersection: An introduction. *Formal Aspects of Computing*, 4(5):470–486, 1992.
- [48] Pramod Joisha and Prithviraj Banerjee. Lattice-based type determination in MATLAB, with an emphasis on handling type incorrect programs. Technical Report CPDC-TR-2001-03-001, Center for Parallel and Distributed Computing, Department of Electrical and Computer Engineering, Northwestern University, March 2001.
- [49] Pramod Joisha and Prithviraj Banerjee. Implementing an array shape inference system for MATLAB. Technical Report CPDC-TR-2002-10-003, Center for Parallel and Distributed Computing, Department of Electrical and Computer Engineering, Northwestern University, October 2002.
- [50] Pramod Joisha and Prithviraj Banerjee. MAGICA: A software tool for inferring types in MATLAB. Technical Report CPDC-TR-2002-10-004, Center for Parallel and Distributed Computing, Department of Electrical and Computer Engineering, Northwestern University, October 2002.



- [51] Pramod Joisha, U. Nagaraj Shenoy, and Prithviraj Banerjee. An approach to array shape determination in MATLAB. Technical Report CPDC-TR-2000-10-010, Center for Parallel and Distributed Computing, Department of Electrical and Computer Engineering, Northwestern University, October 2000.
- [52] Pramod G. Joisha and Prithviraj Banerjee. Correctly detecting intrinsic type errors in typeless languages such as MATLAB. In *SIGAPL APL Quote Quad*, volume 31:2, pages 7–21, New York, NY, USA, 2000. ACM.
- [53] Mackale Joyner. *Array Optimizations for High Productivity Programming Languages*. PhD thesis, Department of Computer Science, Rice University, 2008.
- [54] John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. *J. ACM*, 23(1):158–171, 1976.
- [55] Marc Kaplan and Jeffrey Ullman. A scheme for the automatic inference of variable types. *Journal of the ACM*, 27(1):128–145, January 1980.
- [56] Ken Kennedy. Telescoping languages: A compiler strategy for implementation of high-level domain-specific programming systems. In *IPDPS '00: Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, Cancun, Mexico, 2000. IEEE Computer Society.
- [57] 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.
- [58] A. J. Kfoury and J. B. Wells. Principality and type inference for intersection types using expansion variables. *Theoretical Computer Science*, 311(1-3):1–70, 2004.

- [59] The Mathworks, Inc. *MATLAB Compiler The Language of Technical Computing Users Guide*.
- [60] The Mathworks, Inc. *MATLAB Compiler*, 1995.
- [61] Matrix market. <http://math.nist.gov/MatrixMarket/>.
- [62] Cheryl McCosh. Type-based specialization in a telescoping compiler for ARPACK. Master's thesis, Department of Computer Science, Rice University, Houston, Texas, 2002.
- [63] Vijay Menon and Keshav Pingali. A case for source-level transformations in MATLAB. In *PLAN '99: Proceedings of the 2nd conference on Domain-specific languages*, pages 53–65, Austin, Texas, United States, 1999. ACM.
- [64] Rickard E. Faith Lars S. Nyland and Jan F. Prins. Khepera: A system for rapid implementation of domain specific languages. In *Proceedings of the Conference on Domain-Specific Languages*, pages 243–255, Berkeley, CA, USA, 1997. USENIX.
- [65] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In *OOPSLA '91: Conference proceedings on Object-oriented programming systems, languages, and applications*, pages 146–161, Phoenix, Arizona, 1991.
- [66] Benjamin Pierce. *Types and Programming Languages*. The MIT Press, Cambridge, MA, London, England, 2002.
- [67] Benjamin C. Pierce. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, Department of Computer Science, Carnegie Mellon University, 1991.
- [68] John Plevyak and Andrew A. Chien. Precise concrete type inference for object-oriented languages. In *Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications*, pages 324–340, Portland, Oregon, 1994.

- [69] Markus Pschel, Jos M. F. Moura, Bryan Singer, Jianxin Xiong, Jeremy Johnson, David Padua, Manuela Veloso, and Robert W. Johnson. Spiral: A generator for platform-adapted libraries of signal processing algorithms. *International Journal of High Performance Computing and Applications*, 18:21–45, 2004.
- [70] M. Quinn, A. Malishevsky, N. Seelam, and Y. Zhao. Preliminary results from a parallel MATLAB compiler. In *International Parallel Processing Symposium*, pages 81–87, Orlando, Florida, 1998.
- [71] John C. Reynolds. Design of the programming language FORSYTHE. *ALGOL-like Languages, Volume 1*, pages 173–233, 1997.
- [72] Luiz De Rose 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.
- [73] Markus Schordan and Daniel Quinlan. A source-to-source architecture for user-defined optimizations. In *Joint Modular Languages Conference (JMLC'03)*, volume 2789 of *Lecture Notes in Computer Science*, pages 214–223, 2003.
- [74] Dan C. Sorensen, Richard B. Lehoucq, and Chao Yang. *ARPACK Users' Guide: Solution of Large Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*, 1997.
- [75] Norihisa Suzuki. Inferring types in Smalltalk. In *POPL '81: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 187–199, Williamsburg, Virginia, 1981.
- [76] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1:146–160, 1972.
- [77] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.

- [78] Mark T. Vandevorde. Exploiting specifications to improve program performance. Technical report, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, USA, 1993.
- [79] Todd L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995. Reprinted in *C++ Gems*, ed. Stanley Lippman.
- [80] Todd L. Veldhuizen and Dennis Gannon. Active libraries: Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98)*, Philadelphia, PA, USA, 1998. SIAM.
- [81] R. Clint Whaley and Jack J. Dongarra. Automatically Tuned Linear Algebra Software. In *Proceedings of SC: High Performance Networking and Computing Conference*, Orlando, Florida, November 1998.
- [82] 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, Montreal, Canada, June 1998.