RICE UNIVERSITY

Telescoping MATLAB for DSP Applications

by

Arun Chauhan

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIRMENTS FOR THE DEGREE

Doctor of Philosophy

Approved, Thesis Committee:

Ken Kennedy, John and Ann Doerr University Professor, Chair
Computer Science

Keith D. Cooper, Professor
Computer Science

Behnaam Aazhang, Professor
Electrical and Computer Engineering

Houston, Texas
December, 2003

Abstract

Telescoping MATLAB for DSP Applications

by

Arun Chauhan

This dissertation designs and implements a prototype MATLAB compiler for Digital Signal Processing (DSP) libraries, based on a novel approach called *telescoping languages* for compiling high-level languages. The thesis of this work is that it is possible to *effectively* and *efficiently* compile DSP libraries written in MATLAB using the telescoping languages approach that aims to automatically develop domain-specific application development environments based on component libraries for high performance computing. Initial studies on DSP applications demonstrated that the approach was promising. During this study two new techniques, procedure strength reduction and procedure vectorization, were developed.

In a joint work, a new approach to MATLAB type inference was developed. The inferred type information can be used to specialize MATLAB libraries and generate code in C or Fortran.

A new technique to engineer the optimizing compiler emerged during the course of the compiler development. This technique allows the optimizations of interest to be expressed in an XML-based language and the optimizer in the compiler to be a light-weight specialization engine.

The type inference engine and type-based specialization were evaluated on a set of DSP procedures that constitute an informal library used by researchers in the Electrical and Computer Engineering department at Rice. The evaluation validated the effectiveness of the library generation strategy driven by specialization.

Acknowledgements

In this precarious and long-winding journey of obtaining a doctorate I was guided, encouraged, and supported by several with whom I shared the neighborhood of space and time.

Ken Kennedy has been the perfect advisor to me. Providing just the right amount of guidance, subtly nudging me onto the right course whenever I started to stray, and always willing to take time out of his incredible schedule, Ken has helped me reach a level of maturity that would have been impossible to achieve without his advice. His astonishing amount of acumen in research and a gentle, but firm, personality is surely an inspiration to many who come into his contact.

I am grateful to my committee members for bearing with me through my early draft of the dissertation and accommodating me at short notices. Behnaam Aazhang was kind enough to meet me off campus to sign my documents when he was away for his Sabbatical. I would like to particularly thank Keith Cooper for spending some of his vacation time to read my dissertation and providing several insightful comments. His feedback forced me to see things from perspectives that I had missed and plugged many holes in my arguments.

John Mellor-Crummey would have been my thesis committee member, except that I never got a chance to collaborate with him for any extended period of time. Even without being a committee member he has been a constant source of neat ideas, especially those pertaining to implementation in which he is highly experienced and knowledgeable.

Another person who almost became my thesis committee member is Kathleen Knobe. I spent two great summers with her as her intern at Hewlett Packard's

Cambridge Research Lab and continued the collaboration afterwards. Kath has an amazing capacity to come up with novel ideas. Unfortunately, that work did not become a part of my thesis, but she has been a very important partner in my journey toward becoming an independent researcher.

When Cheryl McCosh started working on the telescoping languages project she and I started a collaboration that lasted until I graduated. I greatly enjoyed working with Cheryl on solving the type inference problem that later became the central piece of her Masters thesis. It was a unique experience to closely collaborate with a motivated and intelligent fellow graduate student that is, unfortunately, not easily available to most PhD students. I believe that we both came out of this collaboration not only technically more informed, having learned from each other, but also more experienced in dealing with the dynamics of a close research collaboration. Cheryl has been a good friend and has taught me several fine points of the American culture and the English language.

Mike Fagan helped me understand the theoretical aspects of type inference, Randy Allen provided several pieces of DSP code in "shrink-wrapped" form, and Walid Taha has been a great person to talk to about multi-staging as well as career path. I want to thank all of them for providing the help I asked for, and more. I would like to thank all the members in the parallelizing compilers group, some of whom have already left, for providing a vibrant working environment including, Vikram Adve, Bradley Broom, Daniel Chavarria-Miranda, Cristian Coarfa, Chen Ding, Yuri Dotsenko, Rob Fowler, Richard Hanson, Chuck Koelbel, Dejan Mircevsky, Qing Yi, Kai Zhang, and Yuan Zhao. Thanks to the support staff for being extremely polite and always ready to help, including Rhonda Guajardo, Donna Jares, Iva Jean Jorgensen, and Darnell Price. Several people in the ECE department provided me with their DSP code and

sometimes sat with me to explain their applications. Thanks to Srikrishna Bhashyam, Vikram Chandrasekhar, Hyeokho Choi, Bryan Jones, Ramesh Neelamani, Dinesh Rajan, Sridhar Rajgopal, Vinay Ribeiro, Justin Romberg, Shriram Sarvotham, Mani Bhadra Vaya, Vidya Venkataraman, and Yi Wan. Thanks also to Joseph Cavallaro who took time out to attend some of the meetings on telescoping languages.

Special thanks are due to my parents for their limitless patience and unflinching support throughout the long years of my PhD. They encouraged me at those times of self-doubt that I underwent upon giving up a cushy job to start a tough life as a poor grad student. I will be eternally indebted to my parents for teaching me to question every phenomenon and every surrounding object with a scientific curiosity. They took active interest in arousing an analytical spirit in me. They tolerated my young-age eccentricities when I turned my bedroom into a physics and chemistry lab and always provided me with whatever I asked for. It is no exaggeration to say that I would not be here without the upbringing I had. I hope that I have fulfilled a part of their dream too and I have not let them down in any way.

These years at Rice are very special to me since this is the time I met Minaxi, whom I married while still pursuing my PhD. For this reason Rice, as well as Houston, will forever have a special place in my heart. Through her love, patience, care, suggestions, and prodding, Minaxi saw me through the ups and downs of the grad life. She rejoiced with me in my moments of discovery and encouraged me in my moments of despair. We had our share of disappointments and frustrations, but Minaxi has been ever the optimist. Her knack for finding ways out of the most difficult situations has always impressed me. I am truly lucky to have Minaxi as my wife and I wish I can play an equally important role in seeing her through her own PhD in Computer Science at Georgia Tech.

Contents

List of Figures

Chapter 1

Introduction

Perfection is achieved not when there is nothing more to be added but when there is nothing left to take away.

–Antoine de Saint-Exupery

High performance computing is at the core of much scientific and technological research. Unfortunately, the community of scientists and engineers has been beset with the dual problem of a shortage of software skills and the difficulty of programming increasingly complex computers for high performance. The problem has been exacerbated by the increasing number of applications demanding high performance and rapidly changing computer architecture. The predominant practical solution so far has been to spend a large amount of time in manually tuning the applications to specific architecture, and then starting all over again when the hardware is updated. This has made the activity of programming for high performance a highly specialized one. As a consequence, either specialized programmers must be hired or the scientists and engineers interested in the applications must divert themselves from their primary job and spend time in tuning the applications. The former is an increasingly

difficult proposition and the latter has the undesirable side-effect of slowing down the progress of science and technology.

One possible solution is to make the end-users *effective* and *efficient* programmers. In order to achieve this not only must the users be able to write programs effortlessly but those programs must also be able to deliver high enough performance that the effort of having to tune the programs becomes redundant. Advanced compiler technology is crucial to realizing this vision.

1.1 Motivation

Users of high performance computing are scientists and engineers who are analytically oriented and well familiar with mathematics and the concept of programming. However, they seek easy to use high-level languages that can capture their algorithms or simulations at a sufficiently abstract level. This is amply demonstrated by the huge popularity of systems like MATLAB®—MathWorks Inc.'s web-site states that there were 500,000 licenses for MATLAB worldwide at the end of 2003. Other examples of highly popular high-level languages include Perl, Tcl/Tk, Mathematica®, Python, etc. Some of the characteristic features of these languages are very high-level operations, high level of expressiveness, and domain specific libraries. For example, MATLAB has several "toolboxes" that are libraries for a variety of scientific and engineering domains and constitute a big reason for MATLAB's popularity.

Unfortunately, programming in MATLAB—in most such high-level languages—comes at the price of performance, which might be measured in terms of space, time,

®MATLAB is a registered trademark of MathWorks Inc.
®Mathematica is a registered trademark of Wolfram Research

or a combination of the two. Engineers typically use the MATLAB environment for early development and debugging of their algorithms and then rewrite their programs in more traditional languages, such as C or Fortran, in order to carry out realistic simulations or for production-level releases. Thus, attractive high-level languages exist today. What is missing is key compiler technology that could bridge the performance gap and make these high-level languages attractive beyond prototyping.

1.2 The Telescoping Languages Approach

Telescoping languages is a novel compilation strategy to automatically develop domain-specific application development environments based on component libraries for high performance computing [40]. In a nutshell, the strategy involves extensively pre-compiling annotated libraries for a variety of calling contexts and selecting the best pre-compiled version at the time of compiling the user programs, which are also referred to as the user *scripts*. The process of selecting the best variants for library procedures is likely to be much faster than compilation of the libraries for the context of the user scripts. If the speculative pre-compilation of libraries has been carried out for carefully selected contexts the object program may be able to reap the benefits of extensive optimizations to a large extent while allowing for rapid compilation of the user programs.

A premise of the telescoping languages approach is that domain-specific applications written in high-level languages rely heavily upon libraries. Indeed, most operations are high-level and directly translate to library calls. This means that any compilation framework that attempts to generate high performance code must pay attention to the libraries.

Past approaches have been based on compiling whole programs, consisting of the top-level user program along with all the library procedures called directly or indirectly from the program. This approach suffers from two distinct disadvantages:

- the compilation time depends on the size of code within the complete call graph, not just the user program; and
- no specialized knowledge about libraries is utilized by the compiler.

Compilation time can become a deterrent when the compiler spends an unexpectedly large amount of time compiling a small user script. An acceptable system should follow the policy of no surprise—the compilation time must be proportional to the size of the program. This suggests that libraries be compiled offline, ahead of script compilation time.

Traditional approaches that compile libraries separately ignore any knowledge about the libraries that the library writers usually have. This knowledge is extremely difficult, if not impossible, to glean from the source code alone. For example, the library writers have a very good idea of the context in which the library procedures are going to be used. This could translate into constraints on input arguments leading to increased opportunities for optimizations. In telescoping languages such properties are captured through an annotation language.

The telescoping languages strategy envisions translating the programs written in the high-level language into an intermediate lower-level language, such as C or Fortran, and manipulating this intermediate representation for high-level transformations. Vendor compilers for the target platform can, then, be used to compile the program into final object code. In this way, highly tuned vendor compilers can be leveraged without having to redo the low-level compiler optimizations that are

already handled very well by these compilers. The *domain library* could be written in the high-level language or directly in the intermediate language. The *language building compiler* extensively analyzes the library based on library annotations. These annotations could consist of constraints on the arguments or return values of library procedures as well as relationships between them. In addition, the library writer may also provide example calling sequences to indicate possible calling contexts.

Armed with this knowledge, the library compiler generates an extensive set of highly optimized versions of the library. This activity can be time consuming, but will be needed only once for a given platform. At this stage, other technologies that optimize for specific platforms could also be employed, for example, ATLAS [58]. The process of deep library analysis can be seen as that of generating an enhanced language on top of the base scripting language, since it has the effect of augmenting the primitive operations in the language with the library procedures. Usually, compilers have very good knowledge of primitive operations and their relative trade-offs, but know little about libraries. The library compiler attempts to bridge this gap. This is the reason it is referred to as the *language building compiler*. One could imagine carrying out this process in a hierarchical manner, enhancing the language in multiple steps. Hence, the term "telescoping languages".

1.3 Hypothesis

This dissertation is based on the hypothesis that it is possible to develop compiler techniques to generate high performance code for numerical applications written in library-based high-level languages. The compilation strategy that enables this is:

effective in that it is able to generate high quality code that is capable of delivering

performance close to what is achievable by an expert programmer in a low-level language; and

efficient in that the strategy generates this high performance code rapidly, in time that is proportional only to the size of the high-level user program.

The proposed strategy to achieve this is *telescoping languages*. The dissertation defends the hypothesis about effectiveness by developing a compiler for libraries written in MATLAB.

1.4 Organization

The rest of the dissertation elaborates on the telescoping languages approach and presents a design of a library compiler. The library compiler is targeted towards compiling DSP library procedures written in MATLAB. However, it is expected that the design and research experience will carry forward to other domains as well.

The first step in compiling MATLAB is inferring variable types. It turns out that specializing the emitted code based on variable types is important in obtaining high performance. Furthermore, extracting accurate type information is crucial in order to be able to generate "good quality" C or Fortran code that could be optimized by the back-end compilers. Chapter 3 defines the type inference problem and describes a new way to solve the problem that is particularly useful in the context of telescoping languages.

There is a whole gamut of optimizations that is available to the compiler writer as a result of the decades of research that has gone into compilation methods. It is natural to ask: which of these program transformation techniques would be useful

from the point of view of optimizing DSP libraries written in MATLAB. A study undertaken to address this issue identified key optimization techniques that result in high performance payoff when applied at the source-level. Chapter 4 describes these techniques. Chapter 5 describes two novel inter-procedural techniques that were developed during the course of this study of DSP applications that also pay off handsomely in specializing DSP libraries.

Chapter 6 discusses a novel approach to engineering the telescoping compiler that emerged in the course of implementing the library compiler. Specialization plays a key role in the telescoping languages approach. It turns out that specialization can also be used as a driver process to implement the optimizations that are found to be relevant for compiling DSP libraries. The optimizations are specified in an XML-based language and the optimizer in the library compiler is simply a light-weight process that reads the external specifications and applies them to a given piece of code. This approach results in substantial software engineering benefits.

Finally, Chapter 8 presents experimental results and Chapter 11 concludes that for the DSP procedures that were studied, given the right forms of specialization transformations and guiding annotations by the library writers, it is possible to achieve substantially higher performance by using the telescoping strategy for library generation than the original code running under the MATLAB environment or naïvely translated into C or Fortran.

Chapter 2

Telescoping Languages

> *It was our belief that if Fortran, during its first months, were to translate any reasonable 'scientific' source program into an object program only half as fast as its hand-coded counterpart, then acceptance of our system would be in serious danger... I believe that had we failed to produce efficient programs, the widespread use of languages like Fortran would have been seriously delayed.*
>
> –John Backus

There are two highly desirable features of a compiler that can play a critical role in determining its acceptance—its *efficiency* and its *effectiveness*. An efficient compiler performs its task quickly and an effective compiler produces object code that performs well. These are opposing goals to implement in a practical compiler and the failure to reach an engineering solution has been a major deterrent against the use of high-level languages beyond prototyping. From an end-user's perspective, an inefficient compiler that takes hours to compile a small program is also ineffective.

Telescoping languages is a novel approach that promises to deliver efficiency and effectiveness to compiling high-level languages. "High-level languages", in this con-

text, refers to domain-specific scripting languages (such as, MATLAB, R, Mathematica, etc.) that are particularly relevant to scientific computation. Such high-level languages will be referred to as *scripting languages* henceforth.

2.1 Key Observation

A scripting language usually makes high-level operations available as primitive operations in the language. For example, in MATLAB, a user can write `a * b` to represent multiplication of two matrices `a` and `b`. Such operations cannot be directly mapped to any operations supported by the underlying hardware, except in the rare cases of compiling a vector operation for vector machines. In most cases, these operations are mapped to calls to procedures within the runtime library that supports the language. If the script is interpreted, the operation triggers a call to the relevant library procedure. If the script is compiled, the compiler generates a call to the appropriate library procedure. From this perspective, it is the language runtime library that really defines the language. Indeed, the library *is* the language! The operators are simply syntactic sugar for ease of programming.

Additionally, a big attraction of the scripting languages is the availability of domain specific libraries. MATLAB has a number of available "toolboxes" for various domains, such as, digital signal processing, fluid dynamics, communications, etc. These toolboxes, or domain-specific libraries, can be viewed as enhancing the basic runtime library that supports the primitive operations in MATLAB.

Libraries, therefore, are at the heart of any computation performed in a program encoded in a scripting language. If a compiler has to optimize such a program it must do a good job of optimizing libraries.

The next question then, is: why are compilers not able to do a good job of compiling libraries? The concept of libraries is an old one and there are numerous compilers and software that support compilation, maintenance, and linking of libraries. Two approaches have been used for handling libraries.

Libraries as Black Boxes

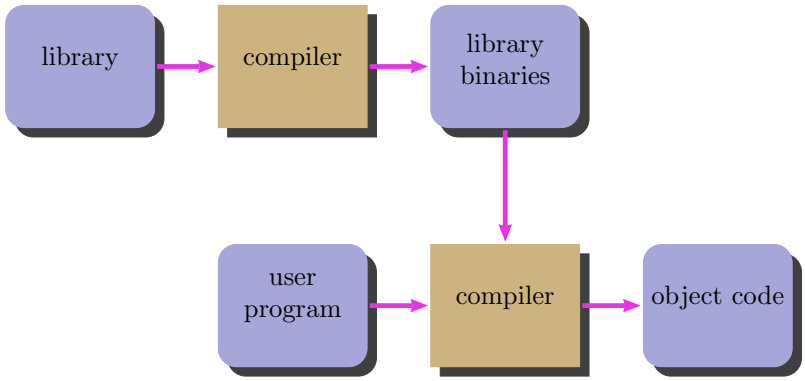


Figure 2.1: Libraries as black-boxes

An overwhelmingly large number of compilers and language systems follow the approach of treating libraries as black boxes. Figure 2.1 depicts this approach. A library is independently compiled and converted into a binary form. The compiler that compiles the end-user program determines which library procedures are used by the user-program, directly or indirectly, and passes the object codes to the linker to link all of those into the final executable code. In this process the end-user compiler has no knowledge of the library beyond the interface it exports, and it does not care about it. The library compiler and the end-user compiler do not cooperate in any way except for adhering to the same interface conventions.

This process is highly efficient. However, it discards all opportunities to optimize libraries that arise when the calling context for a library procedure is known. For example, if the compiler knew that a procedure was called with a certain argument bound to a constant value, it could propagate that constant inter-procedurally. This could expose new optimizing opportunities not only within that procedure but also in the procedures called inside it through a cascading effect.

Whole-Program Compilation

Whole-program compilation is the idea that motivates the other end of the spectrum of compilation approaches that some systems adopt. Figure 2.2 shows this approach. The end-user compiler analyzes the input program to determine all the library procedures that are called by it. It then computes a transitive closure of the call graph by repeatedly adding procedures that are called, directly or indirectly, within those procedures. Once the set of all the procedures has been identified the compiler has precise information of the calling contexts for each procedure and can perform an extensive inter-procedural analysis. This approach is highly effective. However, it is also highly inefficient.

The biggest problem with this approach is the time it takes to compile the user program. It violates a central principle of good software design, which is the principle of least surprise. The compilation time should be proportional to the length of the user program—the user may be greatly surprised if the compiler churns away compiling a hundred-line program for hours simply because the transitive closure of its call graph is huge. There have been compilation systems that attempted to get around this problem by maintaining program repositories, such as the $\mathbf{R}^{\mathrm{n}}$ system at Rice [19].

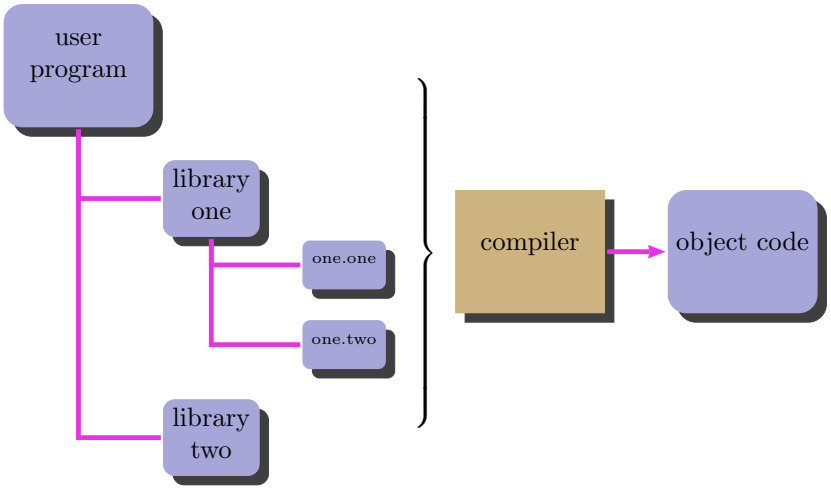


Figure 2.2: Whole-program compilation

Some of the ideas in the $\mathbf{R}^n$ system have motivated the telescoping languages strategy.

A second problem with the approach is that libraries have to be recompiled every time a user program is compiled. This is an enormous duplication of effort. While some of the effort may be mitigated using local caches, it is not entirely eliminated.

Finally, whole-program compilation requires the library sources to be available. This is not always possible, especially when the libraries are carefully guarded intellectual properties. Some efforts in the recent past have tried to address the issue of optimizing whole programs that use libraries in the binary form through link-time optimization [53, 8]. However, these solutions are likely to miss out high-level optimization opportunities that are possible at the source-level.

2.2 The Telescoping Solution

The telescoping languages approach builds on the past work in the compilers community by utilizing several well-known concepts. The approach combines these concepts in a unique way in order to compile high-level scripts for high-performance.

To support the telescoping languages approach, the libraries are analyzed ahead of compiling the user scripts to ensure the efficiency of the script compilation process. In order to retain the benefits of context dependent optimizations, the telescoping languages strategy depends on two primary techniques:

1. hints from the library writers to inform the library compiler about likely contexts; and

2. extensive analysis of the library by the library compiler to generate *specialized variants* that are highly optimized for likely contexts.

Invariably, library writers have an excellent idea of how the library procedures are going to be used. This knowledge is unknown to the compiler at the time of compiling the library since no calling context is available. One way for the library writer to provide this information is through annotations. These annotations would provide the compiler with the possible contexts.

The library compiler uses these annotations to *speculate* about the possible contexts and generates multiple variants, each optimized under the assumption of a speculated context. The script compiler simply chooses the most optimized variant for the actual context in the end-user script.

The annotations do not have to be static and library compilation does not need to be a one-time affair, although it would likely be done in the background and much

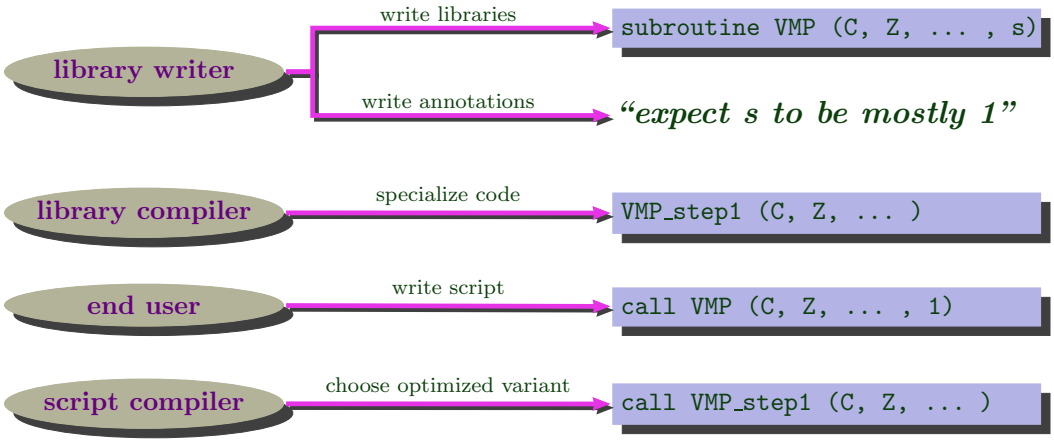


Figure 2.3: Entities involved in telescoping languages

less frequently than script compilation. Library writers may discover new knowledge about the libraries with time. The compiler itself may discover new relevant calling contexts or find some of the existing variants to be not very useful. As a result, the library compiler will need to be run in a maintenance mode from time to time.

Figure 2.3 shows the four entities involved in the telescoping languages approach. In the figure the library writer annotates a hypothetical procedure, `VMP`, to say that a likely calling context for this procedure is one in which the last argument, `s`, is bound to the constant `1`. Based on this knowledge the library compiler generates a specialized variant, `VMP_step1`, that is valid for this specific context and may be better optimized. Other annotations would trigger other specializations. Notice that the library compiler would still need to generate a generic version to be used in case none of the contexts corresponding to the specialized variants match. The script compiler recognizes that there is a specialized variant for the call to `VMP` in the user script and selects the (presumably) more optimized variant.

Notice that the end user writes the script exactly as she or he normally would.

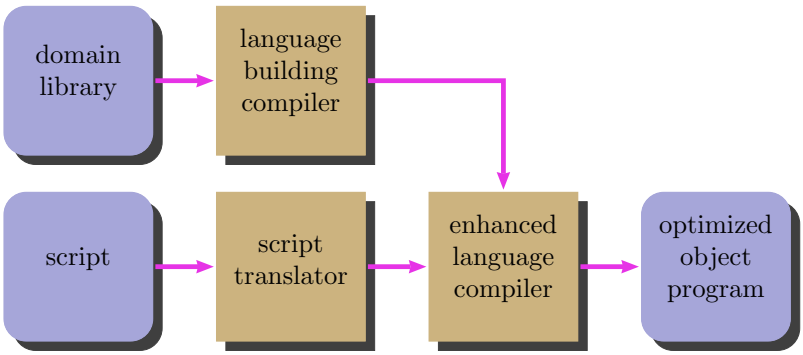


Figure 2.4: Library compilation as language generation

Only the library writer provides the annotations. In addition to, or in lieu of, providing explicit annotations the library writer could also provide example calling programs. In this case the library compiler can perform inter-procedural analysis to characterize the calling contexts automatically and generate optimized variants of the library procedure for those contexts.

While the scripting compiler needs to be efficient and perform in time proportional to the size of the script, the same is not the case for library compiler—the subject of this dissertation. In fact, the library compiler is expected to take a long time in exploring various possible calling contexts for each library procedure and churning out specialized variants based on its analysis and library-writers' annotations. Since library compilation is carried out out only rarely, long compilation time may be acceptable. The process is akin to offline indexing by Internet search engines to support rapid user queries—the telescoping languages approach involves extensive offline library processing to support efficient and effective compilation of end-user scripts.

Recall that for a scripting language the libraries practically define a language.

Therefore, a somewhat more abstract view of telescoping languages is Figure 2.4. Here the library compiler is shown as a "language builder". Extensive analysis of libraries can be seen as enhancing the knowledge of the scripting compiler. Typical compilers have a very good knowledge of the relative trade-offs of primitive operations in a language. Enhancing the knowledge of a script compiler about a library enables the compiler to optimize the calls to procedures in that library the same way as it optimizes primitive operations. Thus, it effectively compiles an enhanced language and can be viewed as an "enhanced language compiler".

Moreover, one can imagine writing more libraries in such an enhanced language and passing them again through a similar process of extensive analysis. This can result in a hierarchy of language levels, while affording the efficiency of the base language. The hierarchical extensibility of languages, without sacrificing performance, is the origin of the term "telescoping".

2.3 Telescoping Languages in Practice

There are two main components of a telescoping languages system: the library compiler and the script compiler. The library compiler interacts with the library writer through a mechanism to direct the library specialization process and creates a database of library variants. The script compiler consults this database in order to optimize the end-user scripts. Several challenges arise.

1. identifying specialization opportunities

2. identifying, and discovering, useful optimizations

3. designing a mechanism for the library writer to provide hints to the library

compiler

4. designing the database to maintain library variants

5. developing an algorithm to utilize the database to optimize scripts as well as higher-level libraries

Each of these challenges pertains to a high-level issue that, in turn, involves solving several lower-level problems.

Specialization Opportunities

Generating highly optimized variants specialized for specific contexts lies at the heart of telescoping languages. Defining the opportunities for specializations and the kinds of specializations to perform is one of the most important issues. The specialization could be based on the input and output arguments of a procedure as well as on the procedure's speculated surrounding context.

Useful Optimizations

The pragmatic approach to implementing a useful telescoping languages system is to identify well-known optimizations that are relevant to compiling scripting languages. It would be desirable to focus on these set of optimizations first. Moreover, the object code emitted by a telescoping compiler is envisioned to be a lower-level language, such as C or Fortran, instead of binary. This allows the telescoping compiler to leverage the highly tuned vendor compilers that are already available for these languages. This also means that a telescoping compiler should focus on those high-level optimizations that the lower-level compilers fail to perform.

Hints by the Library Writer

The next challenge is to design a way to succinctly capture the hints by the library writer while being expressive enough to describe all types of desired optimizations. One highly desirable characteristic of such a mechanism is for it to be language independent, or at least be applicable to a wide range of languages. Moreover, the mechanism has to be simple enough that the library writers will actually use it. Such a mechanism may be realized by way of an annotation language. The language needs to be able to not only describe the call sites based on procedure arguments and return values, but also the surrounding contexts.

There may be more than one way for the library writer to provide hints. For example, in addition to providing annotations, the library writer may find it easier to provide some example calling programs from which the compiler should be able to extract the appropriate context information.

Creating the Variants Database

The task of utilizing the hints by the library writer is a challenge in itself. The process of identifying the context specified by a library writer's annotations or examples may in itself be non-trivial. The database has to be designed in a way that makes lookup, additions (and, perhaps, deletions) of variants easy. It may also be possible to combine contexts together to generate variants for contexts that were not directly specified by the library writer.

Utilizing the Variants Database

Library procedures often call other lower-level libraries. Therefore, utilization of variants takes place at the level of library compilation as well as script compilation. There are likely to be multiple choices for optimizations at every stage. Choosing a particular variant at one call-site may have a cascading effect on other statements. In order to decide between these choices there needs to be a cost model that associates costs with library calls and other operations (such as, array copying) that might accompany them. However, the library compiler can spend much more time exploring different options and performing a "global" optimization, while the script compiler has a limited time in order to be efficient. The process of choosing the right combination of variants poses a complex problem and is likely to involve different trade-offs for script and library compilation.

The design of a variants database and a cost model to utilize that database is out of the scope of this work. The next few chapters discuss solutions to some of the other issues enumerated here.

Chapter 3

Type-based Specialization

> *Not everything that can be counted counts, and not everything that counts can be counted.*
>
> –Albert Einstein

Specialization is a key idea in the telescoping languages strategy. The first question that arises, therefore, is: what kinds of specializations are important for libraries written in scripting languages?

It turns out that one way to specialize libraries written in scripting languages is based on the variable *types*. Most scripting languages are characterized by being typeless or, at best, weakly typed. This affords ease of programming and is, in fact, a major attraction of these languages. However, it also means more work for the compiler if it has to do a good job of optimizing the scripts.

In MATLAB, every variable, by default, is an array. The array could consist of one of the basic types, as shown in Figure 3.1. (A `cell` type is equivalent to a pointer type that could contain any arbitrary type object.) MATLAB is also dynamically typed,

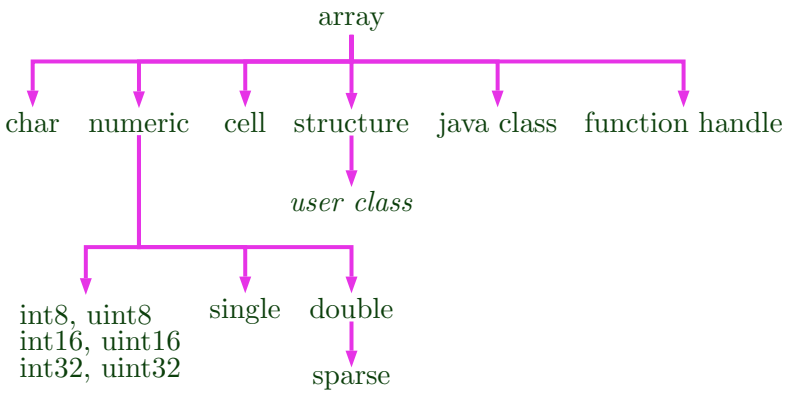


Figure 3.1: Data class hierarchy in matlab

which means that the actual type of a variable depends on the dynamic context at the time the code involving that variable is executed. The operation that is performed on a value can vary depending on its dynamic type. For example, in an expression `a*b`, `*` stands for scalar multiplication, if `a` and `b` are scalars, and for matrix product, if both `a` and `b` are two-dimensional arrays. This is how the MATLAB interpreter works. The MathWorks `mcc` compiler that can emit C or C++, generates code that behaves as the interpreted code would—each variable in the generated code has a generic type and the actual type is inferred at runtime.

In an overwhelming majority of the cases the type of a variable can be completely determined statically. In such cases, it is possible to eliminate the runtime overhead of dynamic inference. This is a well known technique that has been used to optimize high-level languages for years.

Users, who write scientific and engineering—numerical—libraries, utilize the dynamic typing properties of scripting languages to write code that works on a variety of variable types, e.g., `real`, `complex`, `integer`, etc. When used, they are usually called with arguments that all have a consistent set of basic types. For example, if a library

procedure expects two matrix arguments, it might expect both arguments to be either `complex` or `real` but never one to be `real` and the other to be `complex` [14, 13]. The term *type-correlation* is used to describe this behavior of arguments—the arguments are type-correlated if their types have a strong correlation.

Type-correlated arguments present an excellent opportunity to specialize libraries. Instead of relying on dynamic disambiguation of types, specialized variants could be generated for different combinations of argument types, called *type configurations*. Not using the exact type can have drastic performance impact—using a `complex` matrix product, when the values are `real`, may cause the computation to be several times more expensive than it needs to be.

For DSP applications, typically, only one of the variants is what the user desires. It is critical to be able to determine that variant to obtain the best possible performance.

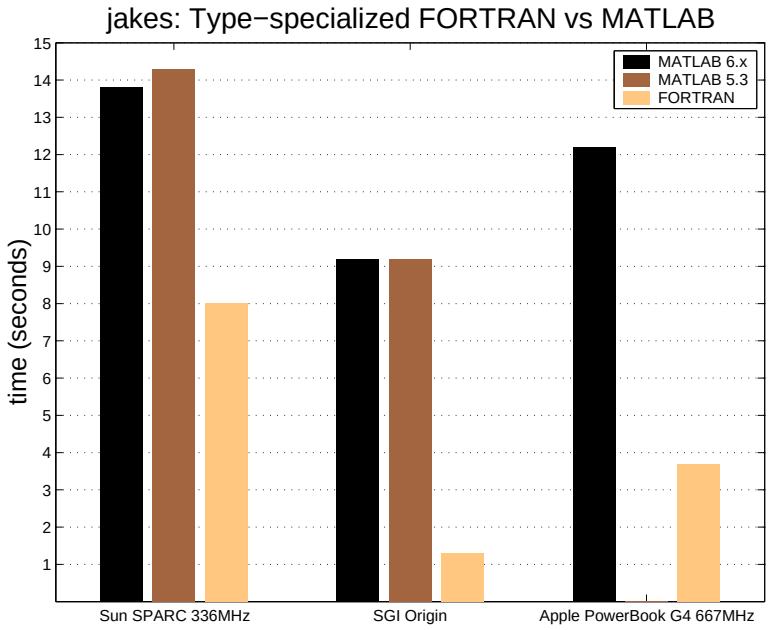


Figure 3.2: Importance of type-based specialization

Figure 3.2 shows the impact of inferring the right types on the performance of a user-level DSP library procedure, called `jakes`. The rightmost bars represent the running time of the procedure with the correct types that were intended by the users for each variable. The left two bars represent complete dynamic inference that is carried out by the MATLAB code. The figure clearly demonstrates that specializing on the right type configuration is very important.

3.1 The Type Inference Problem

Before specializing on type configurations, the key problem to solve is inferring variable types. The existing type inference solutions, including those based on dataflow analyses, determine the "best" possible type for each variable and are often used to prove correctness of the input code. The model used in telescoping languages is that the user develops and debugs the code in an existing interactive environment. The telescoping compiler is then invoked on the debugged code. Therefore, the goal of a telescoping compiler is not to prove correctness, but to determine all possible configurations of variable types that preserve the validity of the code. Each such configuration can lead to a specialized variant.

The types to be inferred depend on the original language (MATLAB) because the meaning of the expressions is language dependent. The inference process also depends on the target language because the set of inferred types must be supported in it. For example, it may not be useful assigning a variable the `logical` type if that type is indistinguishable from integers in the target language.

MATLAB expressions can be of one of the following kinds:

constants `-1`, `2.34`, `-434e-100`, `'const string'`, etc.

identifiers `x`, `y`, `some_identifier`, etc.

arrays `[1,2]`, `[1 2; 3 4]`, `[[a b c]' [2 3 4]']`, etc.

array references `x(5)`, `y(10,8)`, `z((a+b)*c,4,5)`, etc.

cells {`'on';'th'`}, {`4.3,{'on','tw'},[1; 3]`}, etc.

cell references `h{3}`, `h{4,5}`, `g{1,2}{3,4}(1)`, etc.

structure (indirect) `x.y`, `z.a{1,2}{3}.b(4).d`, etc.

The notion of type for the purposes of this work is a tuple $<\tau, \delta, \sigma, \psi>$ associated with each variable where the fields are defined as below. This definition is motivated from that used by de Rose for the FALCON project [26].

τ is the intrinsic type, i.e., `complex`, `real`, `integer`, `logical`, `char`, `struct`, or `cell`.

δ is the dimensionality of the variable; it is 0 for a scalar, equals the number of dimensions for a multi-dimensional array, and equals the number of fields for `struct`.

σ is relevant only for an array and represents a tuple of size δ whose fields are the sizes along all the dimensions.

ψ is relevant only for an array and represents the "structure"[1] of an array, e.g., diagonal, triangular, sparse, etc.

This notion of type is oriented towards MATLAB-like languages with numerical applications in mind where arrays play a primary role in computations and with C

[1]Some researchers prefer the term "pattern" or "layout".

or Fortran as the target language. Newer versions of MATLAB also support object oriented features that are considered out of the scope of this work—most DSP applications do not use those features. Handling of `struct`[2] and `cell`-arrays is described later.

There are two ways to infer the type of a variable from its use:

1. Infer the type of an expression from the types of its constituents. If this expression is assigned to a variable, then the assigned variable gets the inferred type.

2. Infer the type of an expression from its use. For example, if a function or operator argument has a known type then any variable or expression passed as the actual argument must have that type (or a type that can be cast to it). The constituent sub-expressions or variables can be typed if the expression type is known.

The first mechanism involves *forward propagation* of properties and the second would require *backward propagation*.

While Padua and de Rose did not find backward propagation very useful [25], inter-procedural context required for processing telescoping languages can change the situation. In particular, the type of the value(s) returned by a procedure may be only partially known, especially if the input types are also only partially known, since MATLAB procedures and operators are heavily overloaded. This becomes a common case when compiling a procedure whose argument types are unknown so that the values defined in terms of the arguments have only partially known types. As a result, subsequent uses of these values must be examined to narrow down their

[2] MATLAB structure types will always be referred to as `struct` to avoid confusion with ψ.

```
function [ ... ] = codesdhd ( ... )
   det_mf    =  [];
   out_mf    =  [];
   [mfout]   =  match_cor ( ... );
   m         =  coh_cor ( ... );
   dv        =  mfout .* conj(m);
   dvc       =  m .* conj(m);
   out_mf    =  zeros(1, N);
   chan_c    =  zeros(1, N);
   tot_div   =  Tm * (2*Bd - 1);

   for k = 1:tot_div
    out_mf = out_mf + dv(k:tot_div:N*tot_div);
    chan_c = chan_c + dvc(k:tot_div:N*tot_div);
   end
...
```

Figure 3.3: Value of backward propagation

types. Propagating the types backwards have a cascading effect of narrowing down the types of other related values. A similar imprecision that must be resolved by backward propagation of types occurs when some of the procedures being used have unknown type signatures.

To demonstrate the value of backward flow of information consider the DSP code in Figure 3.3. The ellipses (. . .) here are shorthand for code that has been omitted for clarity, rather than MATLAB continuation. In this code, suppose that nothing was known about the procedures `match_cor` and `coh_cor`. Then, the initialization of the variables `out_mf` and `chan_c` proves that the two variables are vectors of size `N`. The variables `dv` and `dvc` are found to be vectors from their uses in indexed expressions in the loop. Moreover, since the operands to a vector addition must match in size, these two variables must be vectors of size at least `N*tot_div`. From here, this information can be propagated backward to the statements that initialized `dv` and `dvc` to infer that

`m` and `mfout` are also vectors of size at least `N*tot_div`. This follows from the fact that operands to element-wise operations must be of matching sizes. Knowing the output types of `match_cor` and `coh_cor` could potentially lead to selecting specialized versions of these procedures for linking. This example illustrating the utility of backward flow of information is, by no means, an isolated one.

The rest of this chapter discusses a solution to the type inference problem in the context of telescoping languages that was developed in joint work with Cheryl McCosh [13]. The details of the static inference component appear in McCosh's Master's thesis [45].

3.2 Static Inference

Static type inference relies on the observation that a statement involving one or more variables imposes certain *constraints* on the types that the variables can have. Inferring the types of variables can be seen as determining an assignment of types to variables such that all the constraints imposed by all the statements are satisfied. The most natural way to express such constraints is using predicates involving type variables over allowable types and equality to relate their values. This can be done for all the four components of the type tuple $<\tau, \delta, \sigma, \psi>$. For example, if τ^v is the type variable that denotes the intrinsic type of a variable `v`, then for a statement, `A = B + C` a constraint could be written for the intrinsic types of `A`, `B`, and `C`:

$$(((\tau^A = \texttt{integer}) \wedge (\tau^B = \texttt{integer}) \wedge (\tau^C = \texttt{integer})) \vee$$
$$((\tau^A = \texttt{real}) \wedge (\tau^B = \texttt{integer}) \wedge (\tau^C = \texttt{integer})) \vee$$
$$((\tau^A = \texttt{complex}) \wedge (\tau^B = \texttt{integer}) \wedge (\tau^C = \texttt{integer})) \vee \ldots)$$

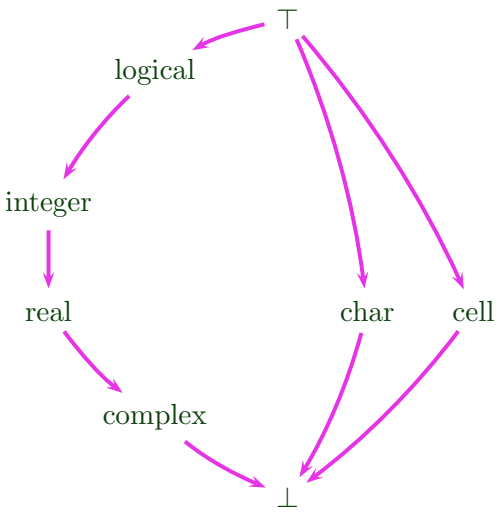


Figure 3.4: Partial ordering for intrinsic type inference

Writing such constraints in terms of only equality and type constants (i.e., `integer`, `real`, `complex`, etc.) is possible, but quickly gets out of hand for overloaded operators like "+". Adding some more relations to this propositional language could simplify the expressions considerably. Suppose that a partial order is defined over the intrinsic types, as shown in Figure 3.4. A type a is lower than b (denoted by $a \prec b$) if an instance of b can be replaced by an instance of a without affecting the correctness of the computation. For example, a variable of type `integer` can be replaced by a variable of type `real`[3]. With respect to such a partial ordering min and max operations can be defined in the obvious manner and the constraints for the "+" operator can be simplified to $\tau^A = \min(\tau^B, \tau^C)$.

In general, such constraints can be written for any arbitrary MATLAB operator or function using predicates connected with Boolean operations. To precisely capture the intrinsic type characteristics of an arbitrary program would require a language powerful enough to model MATLAB (first or higher order logic, for example), which would make type inference extremely difficult. It could be argued, based on structural induction on MATLAB programming constructs, that a simple language based

[3]MATLAB does not perform integer division with the overloaded divide operator. However, in a more general context where dynamic dispatch is possible, a more careful definition of "$\prec$" would be needed.

on Boolean predicates and operations (i.e., propositional logic) is sufficient to capture most common cases accurately and provides an acceptable approximation for others. Then, any algorithm that infers intrinsic types correctly must find an assignment of values to the type variables such that all the constraints for all the program statements are satisfied. Similar constraints can be written for dimensionality, size, and structure.

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

In the presence of arbitrary operators with arbitrary propositional constraints the problem of determining types that satisfy all the constraints is undecidable, in general (halting problem can be easily reduced to it). Even in the more restricted case of straight line code (that has no branches) it is a hard problem. The following theorem states this for array dimensionality.

Theorem 3.2.1 *Inferring feasible (valid) dimensionality combinations for variables in a program without branches is $\mathcal{NP}$-complete.*[4]

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

In order to complete the proof, 3-CNF SAT is reduced to this problem. 3-CNF SAT is the problem of determining whether a satisfying truth assignment to variables

[4] $\mathcal{NP}$-completeness is used here in the sense of algebraic complexity, not bit complexity.

exists for a Boolean expression in conjunctive normal form where each clause consists of exactly three literals. The problem statement in 3-CNF is of the form

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

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

Given a program statement `f(A,B,C)` there are some constraints imposed on the dimensionalities of `A`, `B`, and `C` depending on the function `f`. For example, the function `f` might impose the constraint that if `A` and `B` are both two-dimensional then `C` must also have two dimensions. If the dimensionality of a variable `V` is denoted by δ^V then this translates to the constraint:

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

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

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

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

Suppose that each variable in the program can only have a dimensionality of 0 or 2 (a dimensionality of 0 means the variable is scalar). In that case, the expression $\neg(\delta^v = 2)$ is equivalent to $(\delta^v = 0)$. For every 3-CNF SAT variable v define a program variable `v`. The variable v is assigned *true* iff `v` has two dimensions. Thus, v corresponds to $(\delta^v = 2)$ and $\bar{v}$ corresponds to $\neg(\delta^v = 2)$ or $(\delta^v = 0)$. In order to reduce an instance of 3-CNF SAT into an instance of the problem of inferring feasible dimensionality combinations a program is constructed that consists of a sequence of statements. Each statement corresponds to a clause in 3-CNF SAT and

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

Figure 3.5: Operations, rank constraints and the corresponding 3-CNF clauses

imposes a constraint that corresponds exactly to the clause. Figure 5 defines eight functions and the constraints imposed by each of the functions that correspond to the 3-CNF formula shown in the rightmost column. In a real program these functions could correspond to library procedures that impose the indicated constraints on their arguments.

For each 3-CNF clause of the form $x_1 \vee x_2 \vee x_3$ the program has a statement of the form $\texttt{f}_\texttt{i}\texttt{(x}_1\texttt{,x}_2\texttt{,x}_3\texttt{)}$ where the function $\texttt{f}_\texttt{i}$ is chosen according the table in Figure 5. This construction is polynomial time and log space and the following lemma, which is stated without proof, establishes its correctness:

Lemma 3.2.1 *A given instance of 3-CNF SAT has a satisfying truth assignment iff the variables in the corresponding program have a feasible combination of dimensionalities.*

The proof of the lemma follows directly from the definitions and the construction of the 3-CNF formula. This completes the reduction and hence, the proof of the theorem. □

Theorem 3.2.1 has another implication. Proving a program correct is undecidable in general. However, if the program has no branches then the problem might seem

"easy". The theorem indicates that if array dimensions must be inferred in the absence of any other information then even this can be a hard problem, since for the program to be correct there must be at least one feasible combination of variable dimensionalities.

Corollary 3.2.2 *Proving correctness of linear code (without branches) in the presence of arbitrary operators is $\mathcal{NP}$-hard.*

A similar result holds for intrinsic types and array structure.

Corollary 3.2.3 *Inferring feasible (valid) intrinsic types or array structure for variables in a program without branches is $\mathcal{NP}$-complete.*

Efficient Solution Under Assumptions

Fortunately, certain simplifying assumptions can be made for most practical programs. In particular, assuming a small number of arguments to functions and assuming that everything is defined in terms of a small number of variables (including the input arguments) leads to a polynomial time algorithm.

Each constraint is a disjunction of one or more *clauses*. If p and q are two such clauses out of constraints C_i and C_j then p is said to be *compatible* with q iff there is a value assignment to the constraint variables such that both the clauses can be satisfied. For example, the clauses $\delta^x = 0$ and $\delta^x = 2$ are not compatible, whereas the clauses $\delta^x = 0$ and $\delta^y = 2$ are compatible. If constraints are carefully defined then clauses in a single constraint are always mutually incompatible, i.e., they are mutually exclusive.

Consider an undirected graph, $G = (E, V)$, where E is the set of edges and V is the set of vertices. For each clause c in each constraint in the program add a vertex

v_c to G. Thus, there are as many vertices in the graph as the disjunctive *clauses* in all the constraints in the program. If v_i and $v_j \in V$, then the edge $(v_i, v_j) \in E$ iff clause(v_i) is compatible with clause(v_j). A clique in G exactly defines a configuration of types that would preserve the meaning of the program. Furthermore, the number of cliques is exactly the total number of valid type configurations for the program.

It turns out that under the assumptions of program correctness along with the assumption of a bounded number of arguments to each procedure an algorithm that constructs cliques incrementally is efficient [45]. Once all the cliques have been found the set of constraints induced by each clique can be solved to determine the exact value for each variable type. This approach is used to solve for each of the four components in the type tuple $<\tau, \delta, \sigma, \psi>$ individually.

MATLAB `struct` Type

MATLAB supports C-like structures, i.e., `struct`. A `struct` can have multiple named fields where each field may in turn may be of any arbitrary type. A variable can be a `struct` or a scalar or an array. Scalars and arrays are handled by the type inference mechanism described so far. In order to handle `struct` variables the parser builds an internal representation of the `struct` as a tree and makes appropriate entries in the symbol table. Figure 3.6 shows the tree representation for an example `struct`. In this way, all the internal nodes are resolved during parsing. Each leaf node is uniquely named by the path to it from the root of the `struct` tree and handled exactly as any other program variable. With this small change, MATLAB `struct` types can be handled without any further modifications to the existing type inference approach.

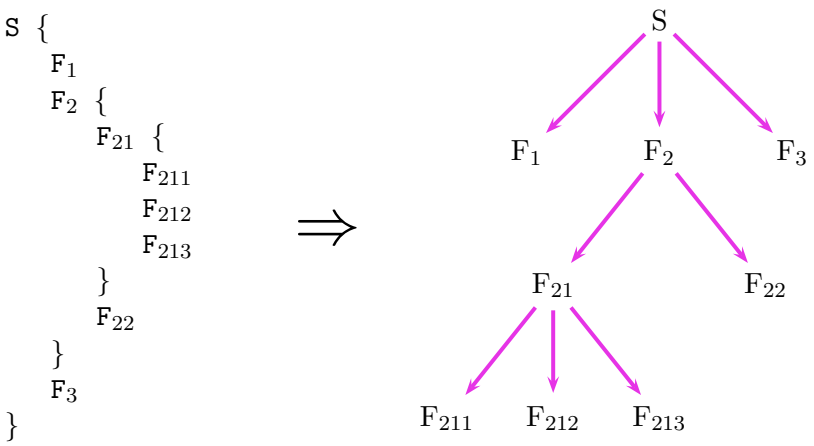


Figure 3.6: Tree representation of matlab `struct`

`Cell`-arrays

`Cell`-arrays in MATLAB are arrays whose elements can be of any arbitrary type. These arrays are dereferenced using braces ({}), thus clearly distinguished from the regular arrays. However, the type of each element of a `cell`-array needs to be inferred separately as it is unrelated to other elements. Since `cell`-arrays can be manipulated just like regular arrays, type-inference needs to deal with the added complexity of inferring the type of each individual element of the array without necessarily knowing the size of the array at compile time. Fortunately, DSP codes rarely use `cell`-arrays that are, therefore, ignored for the purposes of this dissertation. Any inference mechanism that attempts to infer `cell`-array types is likely to need a dynamic inference strategy as a backup.

3.3 Dynamic Inference

Even without `cell`-arrays there are frequently occurring cases in DSP code that are not handled by the static analysis outlined in the previous section. The static analysis does a good job of computing all possible type configurations and taking care of forward and backward propagation of types. However, it has certain limitations.

1. Constraints-based static analysis does not handle array sizes that are defined by indexed array expressions, e.g., by changing the size of an array by indexing past its current extent along any dimension.

2. The strategy in the static-analysis phase to handle control join-points (the ϕ-functions) for intrinsic types and structures is to specialize. For example, in $x_3 = \phi(x_1, x_2)$, if x_1 had the intrinsic type `real` and x_2 had the intrinsic type `complex` then the analysis would double the number of type configurations; x_3 would be `real` in one half and `complex` in the other half. This has the potential of creating a combinatorial explosion of variants.

3. The control join-points are ignored for determining array dimensionality and size. This can lead to a failure in determining an array size statically.

4. Some constraints may contain symbolic expressions involving program variables whose values are not known at compile time.

Dynamic inference of types can complement the static analysis to ameliorate these limitations. It should be emphasized that this is different from disambiguating types dynamically the way MATLAB interpreter does. The dynamic strategy described here carries out the inference at a much coarser level—sometimes only once every time

the procedure is called, or even once per call-site. Therefore, it has a much smaller overhead.

Slice-hoisting for Size Inference

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

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

An example that illustrates the situation that the static analysis fails to handle is shown in Figure 3.7. In this snippet from DSP code arrays `vcos` and `mcos` increase in size around loops. There is no straightforward way to write constraints within the constraints-based approach of Section 3.2 that would accurately capture the sizes of `vcos` and `mcos`. The problem here is that the static analysis ignores ϕ-functions, while they are crucial in this case to determine the sizes. Past studies have shown that array-resizing is an extremely expensive operation and pre-allocating arrays can lead to substantial performance gains [11, 46]. Therefore, even if the array size has to be computed at

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

Figure 3.7: Arrays can be resized in loops

runtime, computing it once at the beginning of the scope where the array is live and allocating the entire array once will be profitable.

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

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

This type of code occurs very often in DSP programs. Notice that the array dimensionality can also be increased by this method, but it is still easily inferred. However, the propositional constraint language used for the static analysis does not allow writing array sizes that change. The only way to handle this within that framework is to rename the array every time there is an assignment to any of its elements, and then later perform a merge before emitting code, to minimize array copying. If an array cannot be merged, then a copy must be inserted. This is the traditional approach to handling arrays in doing SSA renaming [23]. Array SSA could be useful in this approach [41].

Finally, if the value of `N` is not known until run time, such as when it is computed from other unknown symbolic values, then the final expression for the size of `A` will

have symbolic values. Further processing would be needed before this symbolic value could be used to declare `A`.

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

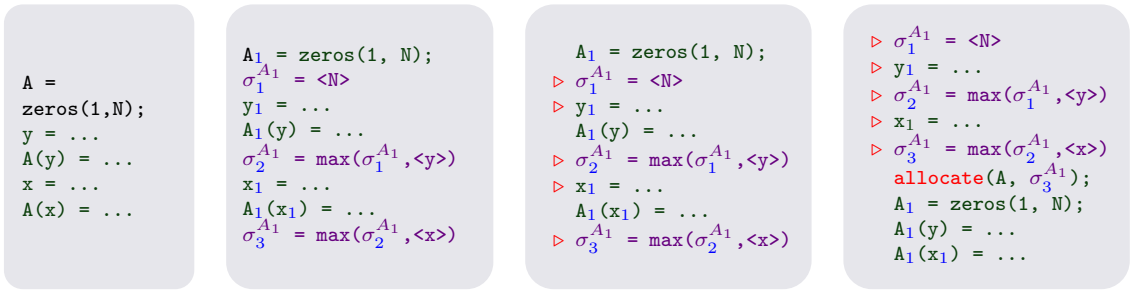
The basic idea behind slice-hoisting is to identify the *slice* of computation that participates in computing the size of an array and *hoist* the slice to before the first use of the array [12]. It suffices to know the size of an array before its first use even if the size cannot be completely computed statically. Once the size is known the array can be allocated either statically, if the size can be computed at compile time, or dynamically. An array size could be determined in one of the following two ways:

- A direct definition defines a new array in terms of the right hand side. Since everything about the right hand side must be known at this statement, the size of the array can be computed in terms of the sizes of the right hand side in most cases.

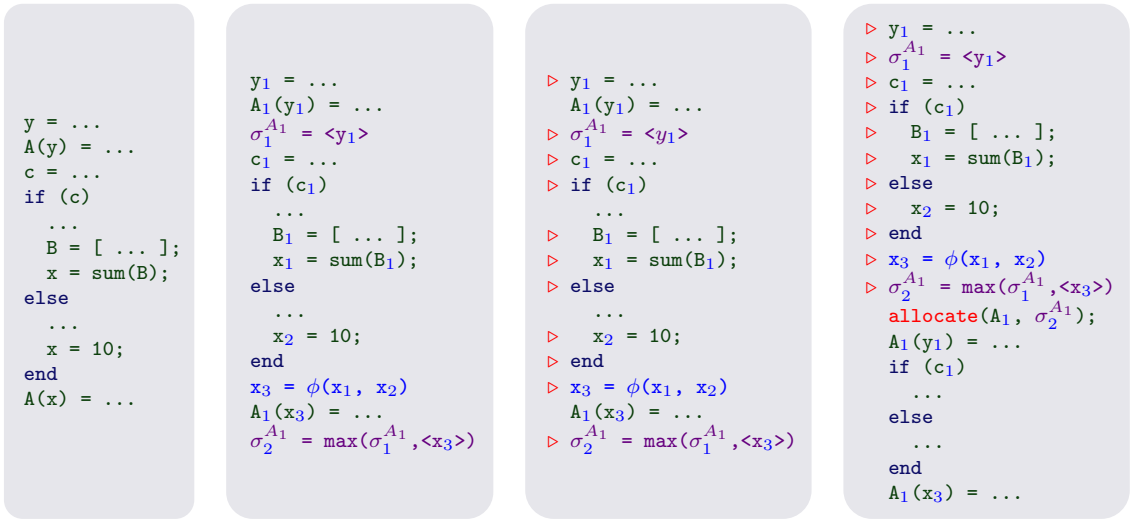
- For an indexed expression on the left hand side, that indexes past the current size, a maximum of the current size and the indices must be computed.

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

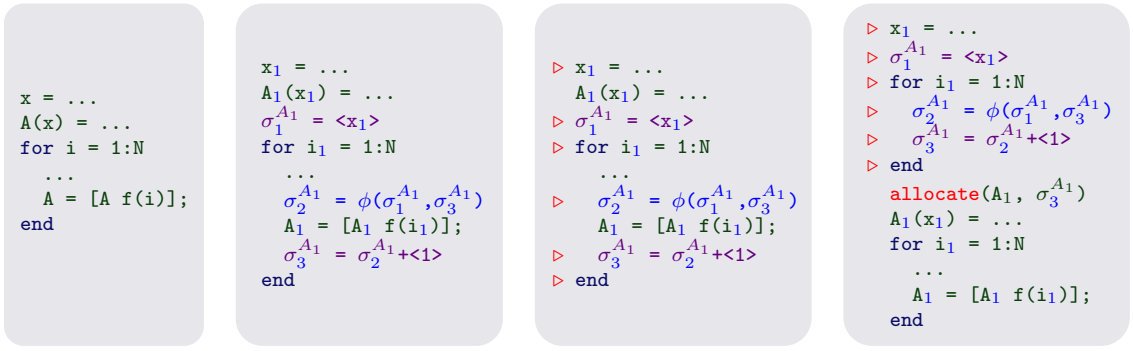
1. transform the given code into SSA form,



(a) Simple case.



(b) Case of a branch.



Initial code | SSA, σ statements | Identifying the slice | Hoisting the slice

(c) Case of a loop.

Figure 3.8: Slice-hoisting

2. insert σ statements and transform these into SSA as well,

3. identify the slice involved in computing the σ values, and

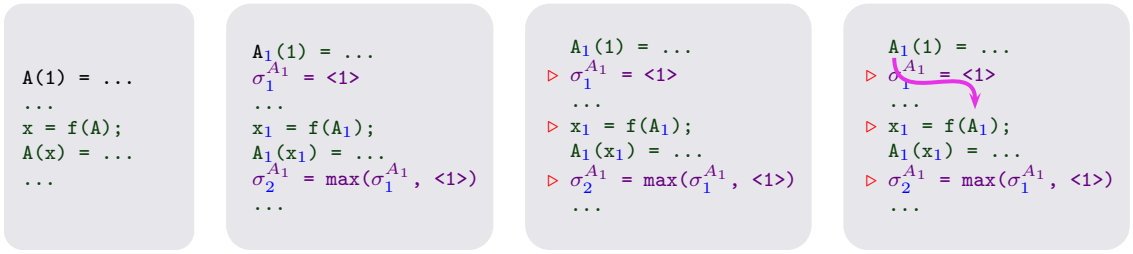
4. hoist the slice.

These steps are illustrated with three examples in Figure 3.8. Steps 1 and 2 have been combined in the figure for clarity. Figure 3.8(a) demonstrates the idea with a simple straight line code. Figure 3.8(b) shows how a branch can be handled. Notice that some of the code inside the branch is part of the slice that computes the size of `A`. Therefore, the branch must be split into two while making sure that the conditional `c` is not recomputed, especially if it can have side-effects. Finally, Figure 3.8(c) illustrates the use of slice-hoisting for a loop. In this case, again, the loop needs to be split. The loop that is hoisted is very simple and induction variable analysis should be able to detect that σ^A is an auxiliary loop induction variable, thereby eliminating the loop. If this is not possible in certain cases, then the split loop reduces to inspector-executor strategy. In this scheme, concatenation to an array, or assignment to an element of the array, does *not* create a new SSA name.

This approach has several advantages:

- It is very simple and fast, requiring only basic SSA analysis. Extraneous ϕ-functions can result in extra (unused) code but never affect the accuracy of the result.

- It can leverage more advanced analyses, if available. For example, advanced dependence analysis can enable slice-hoisting in cases where simple SSA based analysis might fail. Similarly, symbolic analysis can complement the approach by simplifying the hoisted slice.



Initial code | SSA, σ statements | Identifying the slice | Dependence blocks hoisting

Figure 3.9: Dependences can cause slice-hoisting to fail

- Other compiler optimization phases—constant propagation, auxiliary induction variable analysis, invariant code motion, common subexpression elimination—all benefit slice-hoisting without having to modify them in any way.
- It subsumes the inspector-executor style.
- The approach works very well with, and benefits from, the telescoping languages framework. In particular, procedure strength reduction and procedure vectorization[5] can remove certain dependences making it easier to hoist slices.
- Most common cases can be handled without any complicated analysis.

In some cases it may not be possible to hoist the slice before the first use of the array. Figure 3.9 shows an example where a dependence prevents the identified slice from being hoisted before the array's first use. Such cases are likely to occur infrequently. Moreover, a more refined dependence analysis, or procedure specialization (such as procedure strength reduction) can cause such dependences to disappear. When the slice cannot be hoisted the compiler must emit code to resize the array dynamically.

[5]Described in Chapter 5.

```
if (c1)
  % x is real along this branch
  x1 = [1.0 : 0.1 : 100.0];
else
  % x is complex
  x2 = [1.0 : 0.1 : 100.0] + sqrt(-1);
end
x3 = φ(x1, x2);
...
if (c2)
  % y is real along this branch
  y1 = 1::0.1:10;
else
  % y is complex
  y2 = 1:0.1:10 + sqrt(-1);
end
y3 = φ(y1, y2);
...
```

Figure 3.10: Potential explosion of the number of variants

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

Controlling the Number of Variants

Creating a new variant at each ϕ-function for intrinsic type and structure can quickly explode the number of variants in certain cases. Consider the example in Figure 3.10 where SSA renaming has already been carried out. Since conditions `c1` and `c2` are independent of each other the number of variants gets multiplied by four due to the two ϕ-functions. Similar combinatorial explosion can occur for array structure. What is more is that the two get multiplied together.

It may be worth performing a small check for the intrinsic or structure type at runtime. At the control-flow join point the ϕ-function is materialized into a copy operation. In the example of Figure 3.10 two variables will be defined, one assuming that $\texttt{x}_3$ is `real` and other assuming that it is `complex`. The ϕ-function will be replaced by a copy operation within each branch. This results in savings on computations performed on `x` within the "`real`" branch. Alternatively, the ϕ-function could be materialized into maintaining a shadow variable for the intrinsic or structure type of $\texttt{x}_3$ so that its uses may be guarded by checking its dynamic type. In both cases the number of variants is checked at the cost of runtime overheads.

3.4 Comparison with Dataflow Analysis

The strategy presented in this chapter is very different from the dataflow analysis that has traditionally been used to carry out type inference [2]. In particular, the FALCON system uses a dataflow analysis technique to infer MATLAB types [26, 25]. Such dataflow techniques rely on a type lattice over which the dataflow problem is framed. The solution consists of a unique type for each program variable.

For the purposes of telescoping languages, this is inadequate. Since multiple configurations of types can be valid—indeed, intended—all those configurations need to be detected. As seen earlier in Figure 3.2, this is critical for achieving good performance. Moreover, this needs to be done speculatively without the calling contexts necessarily being available.

Dataflow frameworks need to be monotonic over lattices with finite chains for the simple iterative solvers to terminate. Backward propagation, necessary for interprocedural analysis, violates monotonicity, thereby rendering the iterative solvers

unusable in their raw form. In the case of size inference, the lattice also has infinite chains.

The type inference approach based on annotations and constraints, combined with slice-hoisting, offers other unique advantages as well over techniques based on dataflow analysis that are particularly useful in the telescoping languages context.

Enhanced Intrinsic and Structure Types

A consequence of the lattice for intrinsic types (Figure 3.4) is that a `real` variable can be lowered to the `complex` type due to the imprecision of the dataflow analysis. Such an imprecision can be expected because of the inherent approximation that dataflow analysis entails with respect to control flow. This can have serious consequences for the performance of the code.

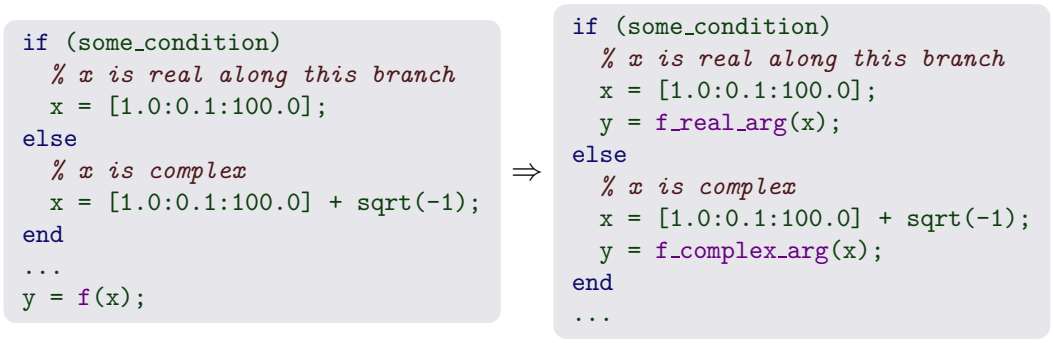


Figure 3.11: Case for enhanced intrinsic types

The left side of Figure 3.11 shows an example where the variable `x` is real along one branch of the conditional and complex along the other. In addition to the imprecision mentioned above this example illustrates another situation that can arise in such cases. In the context of telescoping languages, function calls are likely to have specialized variants. Suppose that the function `f` was specialized for `real` and

`complex` types of its argument. Then, it might be desirable to transform the code to that shown on the right.

In order to detect such an opportunity the intrinsic type of `x` will need to be assigned out of an *enhanced* set of types—one in which variables can have union types. Therefore, the SSA renamed variable for `x` will have the type corresponding to `real` $\cup$ `complex` at the end of the conditional statement. Such union types are very easy to implement in the intrinsic type inference framework of Section 3.2. This would require an additional constraint for each ϕ function such that the intrinsic type of the left hand side is the union of its arguments. In addition, the rules of compatibility change slightly, but everything else remains exactly as before. Whenever a variable has a type that is a union of multiple intrinsic types there is a choice between either generating multiple variants for each of those types or emitting code to do dynamic inference. An enhanced lattice of intrinsic types also exposes the optimizations such as those shown in Figure 3.11. A similar argument can be made for an enhanced lattice for array structure.

Size Inference in the Dataflow World

Size inference as a dataflow problem induces the lattice shown in Figure 3.12. It looks similar to the constant propagation lattice with the important difference that the elements are also linked horizontally. This gives rise to infinite chains leading to potential non-termination of a simple iterative dataflow solver. An example where the lattice is traversed along the horizontal chain by an iterative solver is when the size of a variable changes in each iteration.

One way to get around this problem is to use the basic dataflow formulation and

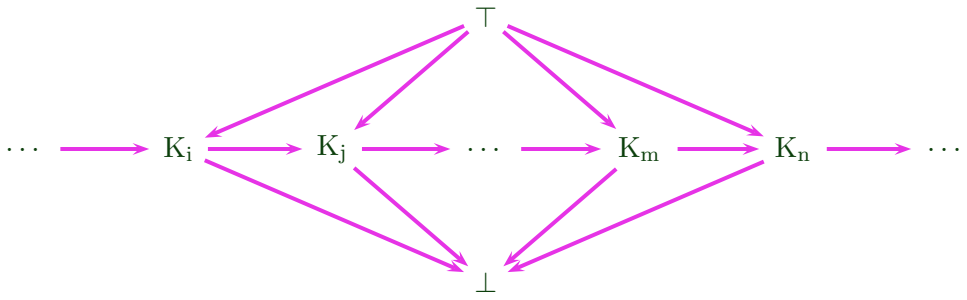


Figure 3.12: Infinite lattice for array size inference

arbitrarily truncate the iterative algorithm in order to ensure termination. When the dataflow solver fails to arrive at a fixed point the size must be inferred dynamically at runtime. This approach, however, would never be able to statically handle cases like those in Figure 3.7 that occur frequently in MATLAB programs.

A common way for DSP programs to resize arrays is to index past their last element, which is handled by slice-hoisting. A dataflow formulation of size inference does not seem to have a simple way to capture this information.

Another approach is to use powerful symbolic analysis to propagate symbolic values and compute maximum values of indexed expressions. This would directly solve the size problem when resizing is done through indices. It can also handle resizing through concatenation inside a loop by propagating statement execution counts and detecting that the size is incremented around the loop in simple steps, thus making it an auxiliary loop induction variable [6]. However, for `while` loops, for which the iteration counts may not be known a priori, a fall-back strategy akin to inspector-executor, will need to be used [24]. Even then, symbolic analysis does not directly solve the problem of an infinite lattice with infinite chains. It only pushes the problem one level away, since symbolic manipulations must deal with the same issue.

By not using dataflow analysis, the combined static and dynamic inference strat-

egy avoids the pitfalls of having to deal with infinite lattice chains and still comes close to symbolic analysis in power.

Chapter 4

Relevant Optimizations

It is a capital mistake to theorize before one has data. Insensibly one begins to twist facts to suit theories, instead of theories to suit facts.
–Sir Arthur Conon Doyle in *A Scandal in Bohemia*

As preliminary research for this dissertation I undertook a study of programs written in MATLAB that were being used for DSP research in the Center for Multimedia Communications (CMC) in the Electrical and Computer Engineering department at Rice. All of these programs consisted of a main script and a number of procedures (MATLAB functions) that were called from the script. For the purposes of this study these procedures were considered to be the user-level library procedures that could be pre-compiled. This assumption was borne out of the fact that many of these procedures were a part of a kind of user-level library and were reused in several different programs.

The purpose of the study was to evaluate the potential of the telescoping languages approach in the context of DSP applications and identify optimizations relevant to

telescoping languages. The study resulted in not only the identification of high-payoff optimizations but also in the discovery of two novel source-to-source inter-procedural optimizations that produced excellent speed-ups in the applications studied [10]. The new optimizations are discussed in the next chapter.

The applications perform various simulations or analyses pertaining to DSP. Digital Signal Processing algorithms tend to make extensive use of linear algebra operations on matrices. The remainder of this chapter describes how the utility of library annotations was confirmed by the study and discusses source-level compiler optimizations techniques that were found to result in high payoff. Experimental evaluation was carried out by performing source-to-source transformation of the applications by hand and are reported in Chapter 8. Performance was measured under the standard MATLAB interpreter.

4.1 Value of Library Annotations

The value of library annotations is demonstrated by the example shown in Figure 4.1. Here, the procedures `change_form_inv` and `change_form` are always called in pairs. Moreover, the value returned from the first is passed on as an argument to the second. The two functions could be combined into one, thus eliminating the overhead of one function call as well as a copy operation. The combination might also enable other optimizations. This is one of the transformations that library annotations aim to uncover.

Specializing procedures based on the annotations provided by the library writer is a major component of the library compiler. The procedures, in the applications that were studied, were found to have great potential to benefit from such annotations. For

```
function [s, r, j_hist] = min_sr1 (xt, h, m, alpha)
  ...
  while ~ok
    ...
    invsr = change_form_inv (sr0, h, m, low_rp);
    big_f = change_form (xt-invsr, h, m);
    ...
    while iter_s < 3*m
      ...
      invdr0 = change_form_inv (sr0, h, m, low_rp);
      sssdr = change_form (invdr0, h, m);
      ...
    end
    ...
    invsr = change_form_inv (sr0, h, m, low_rp);
    big_f = change_form (xt-invsr, h, m);
    ...
    while iter_r < n1*n2
      ...
      invdr0 = change_form_inv (sr0, h, m, low_rp);
      sssdr = change_form (invdr0, h, m);
      ...
    end
    ...
  end
```

Figure 4.1: Annotations can be useful in real-life applications

example, the arguments to function calls frequently have a small range of values (such as, 0 and 1) and this knowledge could potentially uncover optimization opportunities. Many arrays tend to be integers, usually limited to having values -1 and 1. An annotation to this effect could specialize a procedure to operate on the arrays of short integers, or even the `char` type to store small values, resulting in space savings.

4.2 High-payoff Optimizations

There are some well known compiler optimizations that turn out to result in high payoffs for the DSP applications that were studied. *Vectorization* is one such optimization. Vectorization has traditionally been used to transform loops into vector statements to be executed on vector hardware [5, 6]. However, in the context of an interpreted language like MATLAB, this has another implication. Consider the example in Figure 4.2. The loop nest inside the comments is semantically equivalent to the two vector statements immediately preceding it. Even on a scalar machine, replacing the loop nest with the vector statements results in a 33 times speedup on a 336MHz SPARC processor! This is the overhead of calling the library procedures `sin` and `cos` inside a doubly nested loop. The vector statement calls these procedures only

```
function z = jakes_mp1 (blength, speed, bnumber, N_Paths)
....
for k = 1:N_Paths
   ....
   xc = sqrt(2)*cos(omega*t_step*j') ...
      + 2*sum(cos(pi*np/Num_osc).*cos(omega*cos(2*pi*np/N)*t_step.*jp));
   xs = 2*sum(sin(pi*np/Num_osc).*cos(omega*cos(2*pi*np/N)*t_step.*jp));

   % for j = 1 : Num
   %  xc(j) = sqrt(2) * cos (omega * t_step * j);
   %  xs(j) = 0;
   %  for n = 1 : Num_osc
   %     cosine = cos(omega * cos(2 * pi * n / N) * t_step * j);
   %     xc(j) = xc(j) + 2 * cos(pi * n / Num_osc) * cosine;
   %     xs(j) = xs(j) + 2 * sin(pi * n / Num_osc) * cosine;
   %  end
   % end
   ....
end
```

Figure 4.2: DSP applications abound in opportunities for vectorization

once with a matrix argument, instead. In general, library calls have high overheads whenever the library procedures are heavily overloaded. Therefore, loop vectorization can result in tremendous performance improvements. Notice that the scalar version of the loop is much easier to understand and, therefore, the preferred way for the users to write the code.

This example also illustrates the value of *common sub-expression elimination*. Notice that the arguments to `sin` and `cos` are identical expressions. Thus, these need be evaluated only once. Moreover, these are vector expressions, making the gains of common sub-expression elimination even higher. This can be expected to be a typical situation in high-level languages.

Since the functions `sin` and `cos` are invoked on identical arguments, there is an opportunity for replacing these individual calls by a combined call, say `sincos`, which computes both sine and cosine of its argument in less time than the two separate calls. A similar situation was seen in the example of Figure 4.1. These cases can be generalized to replacing a sequence of library calls by another, less expensive, sequence. Such relationships among sequences of library procedures are referred to as *library identities*. The knowledge of library identities is very important for a telescoping compiler to perform high-level optimizations. Library writers are expected to supply these identities, reinforcing the importance of library annotations. Even though identities could also be discovered and recorded by the compiler, library writers' annotations need to provide the base upon which a more extensive knowledge-base could be built. A cost-model of procedure calls would aid the compiler in making a decision about whether or not to carry out a given transformation in a given context.

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

Figure 4.3: Implicit matrix resizing is common in matlab programs

Since DSP applications make heavy use of matrices, they often explore the limits of matrix manipulations in MATLAB. Figure 4.3 shows a DSP code snippet that illustrates a typical example of the way users tend to resize matrices. Naïve code generation will result in huge overheads in allocating and copying the array values. If the array size can be inferred at compile time, either as a constant or symbolically, the entire array can be allocated ahead of the loop. Past studies by Menon and Pingali have shown that significant performance improvements can be achieved, even at the source-to-source level, simply by inserting a `zeros` call before the loop to pre-allocate the array [46].

Another common way to manipulate array accesses found in the applications that were studied, was the use of `reshape`. As the name suggests, this MATLAB primitive prepares an array to be indexed in a manner different from the way it had been accessed before the call. The call allows an array to be *reshaped* into another one that may have different rank or extents.[1] A classic technique called *beating and dragging along* can be used to handle array reshaping [1].

The table in Figure 4.4 summarizes the optimizations that were found to have high-payoffs in this study.

[1] A similar effect can be achieved in Fortran by using `common` blocks.

vectorization	Users prefer writing C or Fortran-like loops, but vectorized statements reduce procedure call overheads and enable other high-level transformations.
common sub-expression elimination	Since many operations operate on arrays, eliminating redundant computation is very important.
constant propagation and constant expression folding	Constant propagation and constant expression folding are known to be useful for traditional languages. These are also useful optimizations for MATLAB and often aid the type inference process.
library identities	Knowledge of the relationships among library procedures is crucial for performing high-level optimizations. Library writers' annotations play a very important role in building this knowledge-base.
array pre-allocation	Pre-allocating arrays eliminates the costly array-copying operations. Array-size inference is critical to be able to do this.
beating and dragging-along	Reshaping arrays is a common occurrence in DSP codes. Appropriate adjustments to the indices of the reshaped array often eliminate the need to copy the array.

Figure 4.4: Summary of high-payoff optimizations

Chapter 5

New Optimizations

If I have seen further it is by standing on the shoulders of giants.
–Isaac Newton in a letter to Robert Hooke

The study of DSP applications led to the discovery of two new inter-procedural optimizations, called *procedure strength reduction* and *procedure vectorization*.

Procedure Strength Reduction Operator strength reduction is a well known compiler optimization technique that replaces an expensive operation inside a loop by a less expensive operation under certain conditions [4]. Procedure strength reduction derives its name from operator strength reduction due to its similarity to the latter in dealing with expensive computations in a loop. Consider a procedure call inside a loop such that some of the actual arguments are loop invariant while the rest are simple expressions involving the loop index (Figure 5.1). It is possible to split the call into an initialization call (f_μ) that performs all the loop invariant computations and an incremental call (f_Δ) that performs the incremental computation. If a substantial amount of computation can be factored

away in f_μ then this can be a beneficial transformation. Section 5.1 describes this transformation in more detail.

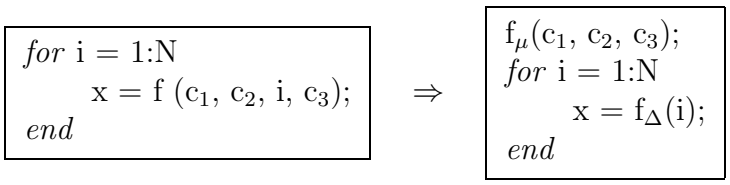


Figure 5.1: Procedure strength reduction

Procedure Vectorization Procedure strength reduction requires that the loop varying arguments be simple expressions of the loop index. However, the function call itself may be part of a dependence cycle in the loop. A reverse situation is possible. Suppose that the function call can be separated from other statements in the loop, even though the expressions in the argument may not be simple. In such a case, after separating the statement into a stand alone loop, it may be possible to interchange the loop and the procedure call to perform loop-embedding [34]. This often makes it possible to vectorize the loop body with respect to the embedded loop, leading to *procedure vectorization*, illustrated in Figure 5.2. This transformation is discussed in more detail in Section 5.2.

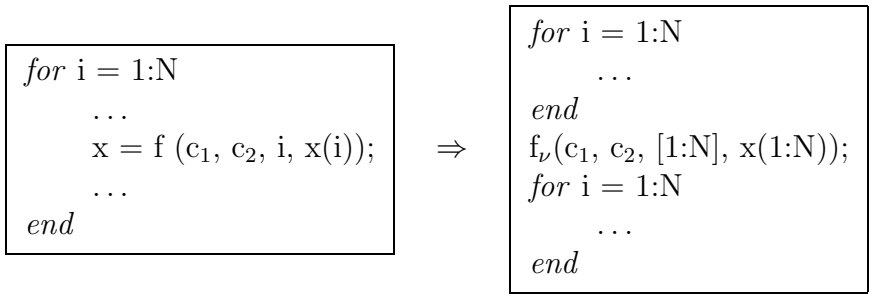


Figure 5.2: Procedure vectorization

It is important to note that both procedure strength reduction and procedure vectorization fit in the telescoping languages framework. Annotations on the library procedures can identify arguments that are expected to be loop invariant. The library compiler can then split or vectorize the procedures that would be subsequently linked into the user script directly by the script compiler.

5.1 Procedure Strength Reduction

It is often the case that library procedures are called in a loop with a number of arguments remaining loop invariant. The computations that depend only on these loop invariant arguments can be abstracted away into an *initialization* part that can be moved outside the loop. The part that is called inside the loop depends on the loop index and performs the *incremental* computation. Figure 5.3 illustrates this idea with a concrete example. The upper part of the figure shows the original code and the lower part shows the code after applying procedure strength reduction. In this case, procedure strength reduction can be applied to *all* the three procedures that are called inside the loops. Only the arguments `ii`, `snr`, and `y` vary across loop iterations. As a result, computations performed on all the remaining arguments can be moved out of the procedures.

Notice that in the case of the procedure `newcodesig` not all the arguments are invariant inside the second level of the enclosing loop nest. Therefore, the initialization part for `newcodesig` cannot be moved completely out of the loop nest. However, procedure strength reduction could be applied again on it resulting in two initialization components—`newcodesig_init_1` and `newcodesig_init_2`.

In principle, for maximal benefit, a procedure should be split into multiple com-

```
% Initialization
....
for ii = 1:200
  chan = jakes_mp1(16500,160,ii,num_paths);
  ....
  for snr = 2:2:4
    ....
    [s,x,ci,h,L,a,y,n0] = newcodesig(NO,l,num_paths,M,snr,chan,sig_pow_paths);
    ....
    [o1,d1,d2,d3,mf,m] = codesdhd(y,a,h,NO,Tm,Bd,M,B,n0);
    ....
  end
end
....
```

```
% Initialization
....
jakes_mp1_init(16500,160,num_paths);
....
[h, L] =  newcodesig_init_1(NO,l,num_paths,M,sig_pow_paths);
m = codesdhd_init(a,h,NO,Tm,Bd,M);
for ii = 1:200
  chan = jakes_mp1_iter(ii);
  ....
  a = newcodesig_init_2(chan);
  ....
  for snr = 2:2:4
    ....
    [s,x,ci,y,n0] = newcodesig_iter(snr);
    [o1,d1,d2,d3,mf] = codesdhd_iter(y);
    ....
  end
end
....
```

Figure 5.3: Applying procedure strength reduction

ponents so that all the invariant computation is moved outside of the loops. In telescoping languages model, `newcodesig` would be a library procedure that would have been specialized for a context in which only its fifth argument varies across invocations. Such a specialization of procedures is context dependent. As mentioned before, example calling sequences and annotations by the library writer would be used

to guide the specialization. An automatic learning process can also be included in the library compiler.

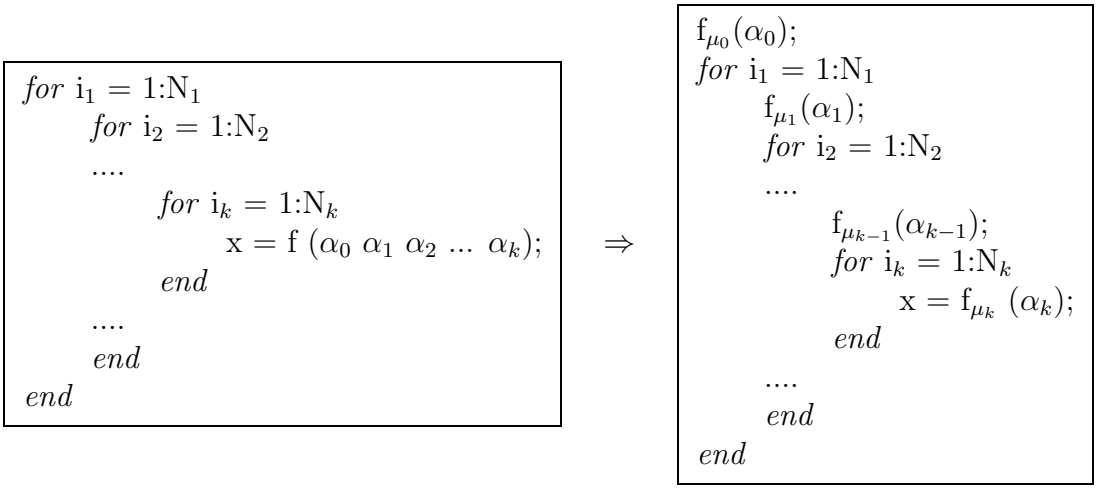


Figure 5.4: The general case of procedure strength reduction

Figure 5.4 shows a general case for multi-level splitting of procedures. In this case α_0 ... α_k are argument sub-sequences where the sub-sequence α_i is invariant at loop level i but not at $i-1$. Indiscriminate specialization can lead to a combinatorial explosion of clones. The extent of reduction in strength must be weighed against the extra overheads of calling the initialization (μ) procedures, the extra space needed to store library clones, and the script compilation time. Appendix A discusses some of the trade-offs involved.

Depending on the arguments that are invariant, one or more of the return values of a procedure may also be invariant. This knowledge is needed to be able to propagate the invariance property. The decision about whether to reduce a procedure in strength may affect the decision for other procedures whose arguments depend on the first procedure.

Procedure and Operator Strength Reduction

In spite of the superficial similarity there are important differences between procedure strength reduction and operator strength reduction. Operator strength reduction replaces an expensive operation in a loop by one occurrence of the operation outside the loop and a less expensive operation inside the loop. Thus, the transformation changes the computation being performed inside the loop and introduces new computation outside the loop.

Procedure strength reduction slices the computations being informed inside a procedure that is called within a loop into those that are loop invariant and those that are not. It is, in a way, inter-procedural loop-invariant code motion. In the telescoping languages framework this slicing is done speculatively at library compilation time. Procedure strength reduction can be combined with operator strength reduction by treating expensive operations inside the procedure as candidates for operator strength reduction. For example, applying operator strength reduction to a multiplication operation inside a procedure f would result in the initializing multiplication operation in f_μ and an incremental addition operation in f_Δ. Such a transformation would require a deeper analysis than simple slicing of the computations inside the procedure.

Procedure Strength Reduction and Automatic Differentiation

Automatic differentiation is a compilation technique to automatically generate code for computing the derivative of a function, given the code that computes the function [31]. The technique relies on the observation that every computation must finally be carried out in terms of either primitive operations in the language or library calls. It uses fundamental principles of differential calculus to transform the given code into

one that computes the derivative. The derivative computation is accurate to within the numerical limits of the underlying platform.

Automatic differentiation could be used to implement procedure strength reduction. Consider a function f that is called with input arguments $p_1, \ldots, p_n$ and returns m values $q_1, \ldots, q_m$. Suppose that automatic differentiation was used to compute $\partial q_i/\partial p_k$, for $1 \leq i \leq n$ and a particular input argument p_k. For every i for which $\partial q_i/\partial p_k$ evaluates to zero the output q_i is independent of the input value p_k. If p_k was a loop index variable then f can be strength reduced using this information. Moreover, under certain conditions, it may be possible to use the derivative to compute the function for a given value of the loop index from the previously computed function values. This could provide an alternative mechanism to generate f_Δ.

5.2 Procedure Vectorization

Vectorization of statements inside loops turns out to be a big win in MATLAB programs, as was established in Chapter 4. This idea can be extended to procedure calls, where a call inside a loop (or a loop nest) can be replaced by a single call to the vectorized version of the procedure. In the context of telescoping languages this can be done by generating an appropriate variant of the procedure.

Consider again the DSP program `ctss` in Figure 5.3. It turns out that the loop enclosing the call to `jakes_mp1` can be distributed around it, thus giving rise to an opportunity to vectorize the procedure. If `jakes_mp1` were to be vectorized, the call to it inside the loop could be moved out as shown in Figure 5.5. This involves adding one more dimension to the return value `chan`.

To effectively apply this optimization in the telescoping languages setting, it must

```
% Initialization
....
chan = jakes_mp1_vectorized(16500,160,[1:200],num_paths);
for ii = 1:200
  ....
  for snr = 2:2:4
    ....
    [s,x,ci,h,L,a,y,n0] = (NO,l,num_paths,M,snr,chan(ii,:,:), sig_pow_paths);
    ....
    [o1,d1,d2,d3,mf,m] = codesdhd(y,a,h,NO,Tm,Bd,M,B,n0);
    ....
  end
end
....
```

Figure 5.5: Applying procedure vectorization to `jakes_mp1`

be possible to distribute loops around the call to the candidate for procedure vectorization. This requires an accurate representation of the load-store side effects to array parameters of the procedure, which would be encapsulated in specialized jump functions that produce an approximation to the patterns accessed. An example of such a representation is a *regular section descriptor* (RSD) [9, 35]. Methods for computing these summaries are well-known.

In practice, the benefit of vectorization will need to be balanced against the cost of a larger memory footprint as well as the costs of specialization as indicated in the previous section.

Loop-embedding and Procedure Vectorization

Procedure vectorization consists of two steps: loop-embedding and vectorization. Loop-embedding is the process of moving the loop surrounding a procedure call into the procedure [34]. It is possible to stop at this stage and still gain significant performance benefits due to reduced procedure call overheads. Procedure vectorization

goes one step further and vectorizes the embedded loop surrounding the original procedure body. This could potentially reduce the applicability of the transformation, but the availability of vectorized versions of most operations in MATLAB allows procedure vectorization to be applicable in many cases. Vectorization further improves the performance gain.

Chapter 6

Engineering a Telescoping Compiler

> *If you want to make an apple pie from scratch, you must first create the universe.*
>
> –Carl Sagan in *Cosmos*

Having identified the type inference engine as the core component that enables the development of the telescoping languages infrastructure the next step is to design and implement this infrastructure. Along with type inference, the focus of this development work is on optimizations that have been identified as being particularly useful for DSP code.

The two main subsystems of the telescoping languages system are the library compiler and the script compiler. The library compiler is responsible for generating and maintaining the variants database and creating procedure constraints in the form of jump functions. The script compiler is a fast compiler that quickly identifies rele-

vant contexts and chooses the most appropriate procedure variants from the variants database. The library compiler also needs to be able to identify contexts and pick variants since libraries are often written hierarchically, with higher-level libraries using lower-level libraries. It is assumed that the lower-level libraries have already been compiled before they are used.

The work in this dissertation focuses only on the library compiler. Those script compilation techniques, such as context recognition, which are also useful in library compilation phase, are addressed. Advanced techniques that ensure *fast* script compilation, and apply cascading transformations, are left for future work.

6.1 Overall Architecture

Procedure specialization plays a pivotal role in the telescoping languages strategy. The importance of type-based specialization in generating effective code in C or Fortran-like intermediate languages has already been established [14]. In the process of developing a library-generating compiler for MATLAB in the telescoping languages context specialization has also emerged as a strategy to implement source-level optimizations. The strategy marks a new way to think about optimizations in a compiler and offers exciting possibilities for the future.

Figure 6.1 shows the overall architecture of the library generating compiler. The compiler accepts a program in MATLAB and emits code in a lower-level language—C or Fortran. The design follows the well known 3-phase architecture of compilers consisting of a parsing front-end, a middle optimizing component, and a code-generating back-end. However, unlike the typical compiler design that entails a fat middle component, the library generating compiler design envisions a light-weight optimizing

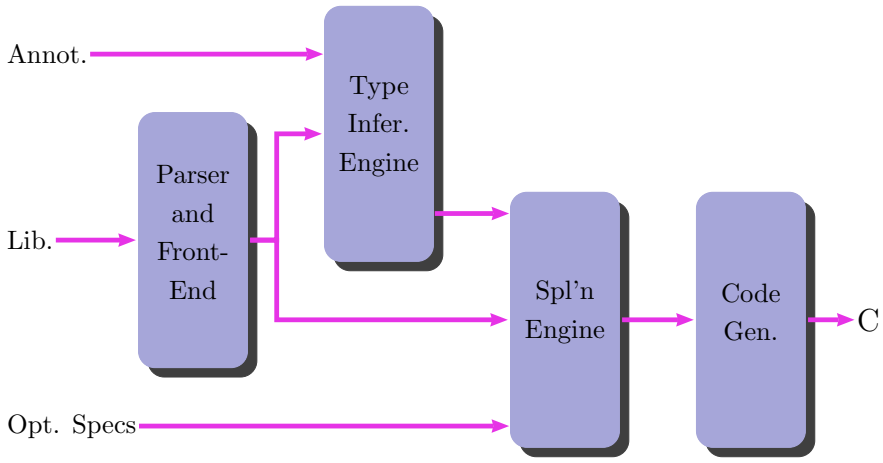


Figure 6.1: Major components of the DSP library compiler

component. The optimizing component is a driver that transforms the programs through a series of specializing transformations. The basic tenet of this design is that most of the source-level optimizations that are relevant to optimizing DSP libraries can be specified as specializations. These optimizations include those identified in Chapters 4 and 5 as well as the type-based specialization described in Chapter 3. The type inference engine is a direct implementation of the type inference approach presented in Chapter 3.

The front-end immediately translates the MATLAB code into an intermediate form in which every primitive operation has been translated into a call to a generic procedure that can handle any operand types. For example, the `+` operand may get translated to a call to a procedure `generic_ADD` that accepts two input arguments and one output argument. Furthermore, complicated expressions are broken down into a sequence of calls to appropriate procedures. This is very similar to what the MathWorks compiler `mcc` does. The specializer operates on this intermediate form to

transform the code and replaces the calls to generic procedures by calls to the specialized variants based on inferred types and the externally specified optimizations. For example, if the operands are proven to be scalar, the call to `generic_ADD` will be replaced by a call to `scalar_ADD` that will get translated to a primitive add operation in the target language by the code generator.

6.2 Optimizations as Specializations

The source-level optimizations that have been identified as high-payoff for MATLAB are:

- loop vectorization
- common sub-expression elimination
- constant propagation
- beating and dragging along
- array pre-allocation

In addition, two new optimizations have been discovered that are also highly relevant:

- procedure strength reduction
- procedure vectorization

Of these, array pre-allocation depends on the type information of a variable. The rest do not require any type information, although they may use it. Array pre-allocation is subsumed in type-based specialization, described in Chapter 3—in fact, that is the primary goal of array-size inference and slice-hoisting.

In order to implement the rest of the optimizations in the library-generating compiler, these optimizations are first expressed as specializing transformations. In other words, the optimization is described in terms of code transformation that can be carried out by replacing parts of the code with the more optimized version, based on dependences, context, and code structure. The optimizer, called the *specialization engine*, reads these *externally specified* optimizations and identifies a piece in the code that satisfies the structural and dependence relations specified in the optimizations. An identified piece is replaced by its equivalent according to the specific transformation and the process is repeated until no more pieces can be matched.

The specialization-based optimization process operates under two assumptions:

1. It is possible to externally specify transformations as code specializations that a specialization engine in the compiler can use.

2. It is always profitable to replace an identified piece of code by its equivalent according to the corresponding specification.

The rest of the chapter proves that the first assumption is reasonable by developing an XML-based language that is shown to be adequate for specifying the interesting optimizations. The chapter also describes a "peep-hole" strategy that the specialization engine can employ to carry out these optimizations.

The second assumption is a simplifying assumption to describe the design of the optimizer, even though there is nothing in the design that depends on it. In order to determine profitability of a transformation the specification language will need to include a cost-metric. However, further investigation will need to be undertaken to study interaction between optimizations and to develop a model that can be used to assess relative costs of alternatives that might exist for a given piece of code.

6.3 An XML Specification Language

The menagerie of optimizations that the telescoping compiler needs to support relies on context-dependent information. Thus, any mechanism that supports specifying these transformations must provide a way to describe contexts. Moreover, most of these optimizations are targeted towards arrays since arrays constitute a very important data structure in high-performance numerical libraries. Thus, the mechanism must provide a way to describe arrays and array sections.

XML, or eXtendible Markup Language, was chosen to describe the transformations for several reasons.

- An XML description is portable, extendible, and standard-based.
- Widely available XML document parsers and unparsers can be leveraged to ease implementation.
- There are generic XML editors available that can be used to facilitate the writing of the specifications. Moreover, the editors need not be restricted to text—they can be extended to let the user write specifications graphically in terms of a control-flow graph, for example.
- An XML specification language makes it very easy to capture the structures of high-level languages without being specific to any language. In other words, as long as the specification is capable of describing suitable structures it is language-neutral.

```
<xsd:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      Optimizations as Specializations.
      Rice University, 2003.
    </xsd:documentation>
  </xsd:annotation>

  <xsd:element name="specializations">
    <!-- unbounded sequence of transformation -->
  </xsd:element>

  <xsd:complexType name="transformation">
    <xsd:complexType>
      <xsd:choice>
        <xsd:sequence>
          <xsd:element name="context" type="preCondition"/>
          <xsd:element name="match" type="stmtList"/>
          <xsd:element name="substitute" type="stmtList"/>
        </xsd:sequence>
        <xsd:sequence>
          <xsd:element name="context" type="preCondition"/>
          <xsd:element name="replaceAllOccurs" type="replacementSpec"/>
        </xsd:sequence>
      </xsd:choice>
    </xsd:complexType>
  </xsd:complexType>

  <xsd:complexType name="stmtList">
    <xsd:sequence minOccurs="0" maxOccurs="unbounded">
      <xsd:choice>
        <xsd:element name="simpleStmt" type="simple"/>
        <xsd:element name="twoWayBranchStmt" type="twoWayBranch"/>
        <xsd:element name="multiWayBranchStmt" type="multiWayBranch"/>
        <xsd:element name="loopStmt" type="loop"/>
        <xsd:element name="anyStmt" type="wildcard"/>
      </xsd:choice>
    </xsd:sequence>
  </xsd:complexType>

</xsd:schema>
```

Figure 6.2: Outline of the XML schema for optimizations as specializations

Syntax

Figure 6.2 shows the high-level outline of the schema for the XML specification language. The complete schema appears in appendix B. The complete set of optimizations are written as a sequence of *specialization rules*. Each rule consists of a *context*, a *match* and a *substitute*. A match or a substitute consists of a sequence of statements each of which can be of any of the four recursively defined types—simple statement, loop, two-way branch, and multi-way branch. The schema has been designed with MATLAB in mind even though it is extendible. The four statements that the schema allows are sufficient to express any MATLAB program structure. The "variable names" allowed under the schema are SSA renamed so that the names refer to values rather than memory locations. The *application* of a rule involves searching for the pattern specified by the match of the rule under the specified context and replacing it with the substitute as long as no dependences are violated.

Figure 6.3 shows an example of a specialization rule for scalar addition. The rule is applicable only when the context is satisfied, which happens when the two operands to the `generic_ADD` procedure are both scalar. Similar rules could be written for specializing `generic_ADD` to matrix-matrix addition, scalar-matrix addition, and so on.

The example illustrated here specializes a single simple statement. In general, an XML description following the given schema can exactly describe a control-flow graph. Indeed, the class hierarchy that is used internally in the front-end of the compiler exactly mirrors the statement structures permissible under the schema. Thus, there is a simple statement class, a loop class, and so on. Effectively, this class hierarchy defines a grammar that can generate a class of control-flow graphs and there

```
<specialization>
  <context>
    <type var="x" dims="0"/>
    <type var="y" dims="0"/>
  </context>
  <match>
    <simpleStmt>
      <function> generic_ADD </function>
      <input> <var>x</var> <var>y</var> </input>
      <output> <var>z</var> </output>
    </simpleStmt>
  </match>
  <substitute>
    <simpleStmt>
      <function> scalar_ADD </function>
      <input> <var>x</var> <var>y</var> </input>
      <output> <var>z</var> </output>
    </simpleStmt>
  </substitute>
</specialization>
```

Figure 6.3: Specialization rule for `generic_ADD` for scalar operands

is an isomorphism between the grammar and the schema. There are three important consequences of this mirroring:

1. Since the internal class-hierarchy is capable of representing the control-flow graph of any arbitrary MATLAB program, it follows that the XML schema has the same power of expression.

2. The control-flow graph is a widely used intermediate representation in compilers. Therefore, describing structures directly in the form of a control-flow graph provides a language-independent way of specifying specializing transformations so that it can be easily used (or extended) for languages other than MATLAB.

3. The isomorphism between the specification language and the grammar of the control-flow graph (defined in terms of the hierarchy of CFG classes) greatly

simplifies the process of recognizing the specified structures in a given program.

Even though the power of the language is illustrated by arguing in terms of the control-flow graph, the language is easier understood in terms of the abstract syntax tree. Since there is a one-to-one correspondence between an abstract syntax tree of a MATLAB program and its control-flow graph in the library compiler, the specializer has a choice of working on either representations.

In addition to the ability to specify a template to match it is often useful to be able to refer to all occurrences of a variable or expression. In lieu of the `<substitute>` element the schema allows writing a `<replaceAllOccurs>` element that specifies replacing all occurrences of a given variable or an array section by another.

As the name suggests the match element specifies a *pattern* that is matched by the specialization engine against the procedure being compiled. Before describing how the pattern is matched, one more syntactic entity must be added to the language to make it sufficiently powerful to handle the relevant optimizations—wild card. Often the match of a rule can tolerate arbitrary intervening statements, which would normally also appear in the substitute. In fact, this is the more common and the more powerful form of the match element. A wild card matches any simple or compound statement and can can match multiple occurrences according to the attributes `minCount` and `maxCount`. The default value for both is one. Each wild card element must also be supplied with a unique integer label that can be used within the substitute element to identify the corresponding match.[1]

[1]Those familiar with Perl might recognize this as a generalization of tagged regular expressions.

Applying the Specialization Rules

In order to apply a rule the specialization engine first converts the match part of the given specification into a control-flow graph. However, this is not a regular program control-flow graph but a *pattern* that must be matched against the control-flow graph of the procedure being compiled.

A specialization rule, $\mathcal{R}$, can be denoted by a three tuple $<C, P, S>$ where,

C is the context that must be satisfied for the rule to be applicable

P is the pattern that must be matched for the rule to be applicable, and

S is the equivalent code that must be substituted for the matched part if the rule is applicable, or a replacement rule that replaces all occurrences of a variable or array section.

Given the abstract syntax tree, T, of the procedure being compiled and a sequence of specialization rules, $\mathcal{R}_1$, $\mathcal{R}_2$, ..., $\mathcal{R}_n$, the specialization engine follows the algorithm in Figure 6.4 by calling the `specializer` procedure for each rule.

An important part of this algorithm is the procedure `search_pattern`. The ground rules for matching include:

- A "variable" in the specification can match any expression tree, including simple variables and constants. However, multiple occurrences of a variable must match exactly the same expression tree. All program variables are assumed SSA-renamed to avoid spurious matchings.

- A "constant" in the specification matches a program constant of the same value.

```
input:
        specialization rule, R =<C, P, S>
        abstract syntax tree, T
output:
        transformed syntax tree, T'
uses:
        search_pattern
        replace_pattern
        replace_occurrences

procedure specializer
    return if the context C not verified
    L = list of the top-level statements in T
    pattern_handle = search_pattern(P, T)
    if found
        if S is a substitute then
            if replacing P by S does not violate any dependencies
                T' = replace_pattern(T, pattern_handle, S)
            else
                T' = T;
            endif
        else
            T' = replace_occurrences(T, pattern_handle, S)
        endif
    endif
    // now repeat the process for each statement recursively
    for each compound statement, M, in L
        H = abstract syntax tree for M
        H' = specializer(R, H)
        T' = T with H replaced by H'
        T = T'
    endfor
    return T'
```

Figure 6.4: The algorithm for the specialization engine

- A "statement wild card" (`<anyStmt>`) can match any simple or compound statement.

It is important to recognize that `search_pattern` can use standard regular expression matching techniques to search for a pattern consisting of entities of the abstract syntax tree.

The remainder of this chapter uses the XML schema described above to write the relevant optimizations as specializations. Only common subexpression elimination cannot be described by the language in its current form as there is no way to write a pattern that can match an expression's occurrences across boundaries of compound statements.

6.4 Specifying the Optimizations

With the XML specification language in hand, the relevant optimizations can now be written down as specializations in terms of this language. This section describes each optimization as a specialization using the XML schema of Figure 6.2. Some other optimizations are also described using the language to illustrate its power.

It should be noted that the specialization engine carries out a transformation only if the transformation will not violate any dependences. Thus, the specializations are more than simple pattern replacement—dependence information is a critical component in applying these specializations.

Type-based and Value-based Specializations

The simplest specializations to specify are type-based specializations. One such specialization was illustrated in the previous section in Figure 6.3. Similar specialization rules can be written for all library procedures for each type configuration for which there is a specialized variant.

The specialization rules can also be written, and even combined with type-based rules, for the contexts that require variables to have certain values. For example, there may be a specialized variant of a numerical library when a stride argument has the value one. This is a common scenario in real libraries. A less frequent situation is when the specialized variant depends on the value of a variable being in a certain range. This situation cannot be described using the current schema, however the schema can be easily extended to allow for it. One issue in handling ranges is that the compiler analysis that needs to be performed to propagate range values is very complicated and the effort spent in doing that may not be worthwhile, at least in the first version of a compiler.

Loop Vectorization

Loop vectorization involves converting a scalar loop into a vector statement. Suppose that a function `f` has a vectorized version called `f_vect`. If `f` takes one input and returns one value, then a specialization rule for vectorizing a single-loop could be written as in Figure 6.5.

In a more readable format, the specification performs the following transformation:

```
<specialization>
  <match>
    <forLoopStmt index="i">
      <lower>L</lower> <upper>U</upper> <step>S</step>
      <body>
        <anyStmt label="1" minCount="0" maxCount="unlimited"/>
        <simpleStmt>
          <function>f</function>
          <input>
            <asection var="a"><dim><lower><var>i</var></lower>
            <upper><var>i/</var></upper></dim></asection>
          </input>
          <output>
              <asection var="b"><dim><lower><var>i</var></lower>
              <upper><var>i/</var></upper></dim></asection>
          </output>
        </simpleStmt>
        <anyStmt label="2" minCount="0" maxCount="unlimited"/>
      </body>
  </match>
  <substitute>
    <simpleStmt>
      <function>f_vect</function>
      <input><asection var="a">
          <dim>
            <lower><var>L</var></lower>
            <upper><var>U/</var></upper>
            <step><var>S/</var></step>
          </dim>
      </asection></input>
      <output><asection var="b">
          <dim>
            <lower><var>L</var></lower>
            <upper><var>U/</var></upper>
            <step><var>S/</var></step>
          </dim>
      </asection></output>
    </simpleStmt>
    <forLoopStmt index="i">
      <lower>L</lower> <upper>U</upper> <step>S</step>
      <body>
        <putStmt label="1"/>
        <putStmt label="2"/>
      </body>
  </substitute>
</specialization>
```

Figure 6.5: Loop-vectorization as specialization

```
for i = L:S:U
   S1*
   b(i) = f(a(i))
   S2*
end
```

⇒

```
b(L:S:U) = f_vect(a(L:S:U))
for i = L:S:U
   S1*
   S2*
end
```

Notice that there may be other statements in the loop that are preserved. The dependence check in the specialization engine ensures that the transformation is legal in a specific case, i.e., no dependences are violated with respect to the statement groups $\mathtt{S}_1$ and $\mathtt{S}_2$.

Even though the XML specification appears complicated, it is, in fact, very simply structured. Moreover, it is an intermediate representation that the library developer or the specialization writer never sees—a front-end editor presents the specification in a graphical or easy-to-edit format. In describing the rest of the transformations self-evident details will be omitted for the sake of clarity.

Beating and Dragging Along

Beating and dragging along in the context of MATLAB is the elimination of the `reshape` primitive by replacing all occurrences of an array section in terms of its new indices induced by the `reshape` call. For example, if a `reshape` call changes the indices of a linear array `A` into a two-dimensional array then all subsequent uses of a section of `A` are rewritten in terms of the original linearized indices. Figure 6.6 shows the XML specification for this specialization rule. It uses the second major construct provided by the XML schema—the `<replaceAllOccurs>` element.

The rule will change if the original array is reshaped into a different number of dimensions. Writing a rule for each possible number of dimensions is not so bad since in real code the array dimensions rarely exceed a small number.

```
<specialization>
  <context>
    <type var="x" dims="2" sizes="m n"/>
  </context>
  <match>
    <!-- a = reshape(b, m, n) -->
  </match>
  <replaceAllOccurs>
    <occurrence>
      <!-- match section a(i, j) -->
    </occurrence>
    <replacement>
      <!-- replace by b(scalarADD(scalarMULT(scalarSUB(j,1),m)),i) -->
    </replacement>
  </replaceAllOccurs>
<specialization>
```

Figure 6.6: Beating and dragging along as specialization

Another primitive function that is a candidate for beating and dragging along is array transpose. Once again, the rules will vary depending on the number of dimensions involved.

Procedure Strength Reduction

Procedure strength reduction is described easily as a template in which the procedure call inside a loop is split into two, as shown in Figure 6.7. The figure illustrates the rule for a function `f` that takes three arguments, one of which is the loop index variable. The specialization engine will not perform the transformation if the other two arguments change within the loop since that would violate the dependence. The specification for a procedure that can be strength reduced multiple times in a loop nest can be written as a straightforward extension of the shown rule.

Recall that the front-end generates an intermediate code in which *every* primi-

```
<specialization>
  <match>
    <forLoopStmt index="i">
      <lower>L</lower> <upper>U</upper> <step>S</step>
      <body>
        <anyStmt label="1" minCount="0" maxCount="unlimited"/>
        <!-- simple statement f(a, b, i) -->
        <anyStmt label="2" minCount="0" maxCount="unlimited"/>
      </body>
    </forLoopStmt>
  </match>
  <substitute>
    <!-- simple statement f_init(a, b) -->
    <forLoopStmt index="i">
      <lower>L</lower> <upper>U</upper> <step>S</step>
      <body>
        <putStmt label="1"/>
        <!-- simple statement f_iter(i) -->
        <putStmt label="2"/>
      </body>
    </forLoopStmt>
  </substitute>
</specialization>
```

Figure 6.7: Procedure strength reduction as specialization

tive operation translates into a procedure call. This has a significant consequence: once the primitive operations have been specialized for scalar operations, procedure strength reduction can be applied to reduce the primitive operations in strength. This is exactly the process of traditional operator strength reduction. Handling operations and library procedures uniformly clearly illustrates how operator strength reduction can be handled as a special case of procedure strength reduction in this context.

Procedure Vectorization

Procedure vectorization follows the pattern of procedure strength reduction. The simple specification for procedure vectorization is shown in Figure 6.8. It is a speci-

fication for a function `f` that has a single argument.

Constant Propagation

Constant propagation can be represented as a specialization with the observation that the language provides a way to detect constants (with the `constant` attribute of the `type` element). Then, the replace-all-occurrences feature can simply replace a variable initialized to a constant by the constant value.

Compile-time Evaluation

One of the most important optimizations performed by compilers is to evaluate as much code as possible at compile time. This has to be done carefully since the com-

```
<specialization>
  <match>
    <forLoopStmt index="i">
      <lower>L</lower> <upper>U</upper> <step>S</step>
      <body>
        <anyStmt label="1" minCount="0" maxCount="unlimited"/>
        <!-- simple statement f(a(i)) -->
        <anyStmt label="2" minCount="0" maxCount="unlimited"/>
      </body>
    </forLoopStmt>
  </match>
  <substitute>
    <!-- simple statement f_vec(a(L:S:U)) -->
    <forLoopStmt index="i">
      <lower>L</lower> <upper>U</upper> <step>S</step>
      <body>
        <putStmt label="1"/>
        <putStmt label="2"/>
      </body>
    </forLoopStmt>
  </substitute>
</specialization>
```

Figure 6.8: Procedure vectorization as specialization

piler must perform the evaluation in the context of the semantics of the language being compiled. If such an evaluation is feasible then it is possible to specify compile-time evaluation using the proposed schema. Consider the common example of constant-expression folding. If two operands to a primitive operation are constant, then it is invariably a good idea to replace the computation by the evaluated result (assuming no exceptions are generated due to the evaluation). The XML specifications will need to be enhanced by defining equivalent compile-time operations for all the targeted operations in the source language. Constant-expression folding for scalar values may be carried out by the back-end compilers for the object code in C or Fortran. However, performing the compile-time evaluation for high-level operations is something the back-end compilers cannot handle; for example, multiplying two constant matrices.

Other Optimizations

There are several other optimizations that are possible within the framework even though these are not high-payoff for compiling DSP libraries written in MATLAB. The table in Figure 6.9 shows some of those optimizations along with the outline of how they can be described using the XML-based specification language.

Out of these, while-for conversion may be a useful one for compiling MATLAB since it is possible to write C-style "for"-loops in MATLAB that are really while-loops. Converting those to equivalent for-loops may be critical in being able to apply other optimizations.

optimization	*match element*	*substitute element*

while-for conversion	`[context: i, L, U are scalar`
	`S is scalar constant]`
	`i = L`
	`statement_group_1`
	`while (i <= U)`
	`  statement_group_2`
	`  i = i + S`
	`  statement_group_3`
	`endwhile`
		`statement_group_1`
		`for i = L:S:U`
		`  statement_group_2`
		`  statement_group_3`
		`endfor`

empty-loop elimination	`for i = L:S:U` `endfor`	`[empty substitution]`

loop-invariant code motion	`for i = L:S:U`
	`  statement_group_1`
	`  x = f(a, b)`
	`  statement_group_2`
	`endfor`
		`x = f(a, b)`
		`for i = L:S:U`
		`  statement_group_1`
		`  statement_group_2`
		`endfor`

copy propagation	`x = y`	`[replace all occurrences]`
		`x`
		`[by]`
		`y`

Figure 6.9: Other examples of optimizations as specializations

6.5 The Integrated Approach

The unified approach to handling the optimizations that are relevant to the telescoping languages has several advantages:

1. New optimizations can be rapidly added to the compiler as long as these can be expressed using the XML-based language. This has tremendous software engineering implications for compiler developers.

2. Optimizations can be applied in different order, and even multiple times, facil-

itating a test-bed to explore interactions among the different optimizations.

3. Most program transformations are carried out by the specialization engine. It is a much smaller piece to debug and verify than separate pieces of code for each optimization, resulting in a greater confidence in the accuracy of the compiler.

4. The design opens up new avenues for research into cost-based, possibly self-learning, program transformation techniques for telescoping languages.

While the design results in an intellectually satisfying integrated architecture for high-level optimizations, not all optimizations may be expressible within the framework of the XML schema presented in this chapter. Moreover, certain properties of MATLAB and the overall architecture of the compiler work together to allow this framework to work.

- The intermediate representation simplifies complicated expressions into individual procedure calls. Therefore, specifying transformations at the granularity of statements turns out to be adequate.

- Every primitive operation is treated as a procedure call. This is essential to do in a weakly-typed high-level language, such as MATLAB, since the meaning of the operation is not pinned down until its operands' types have been inferred. Thinking of every operation as a procedure call makes a uniform treatment of statements possible.

- MATLAB procedures have no side-effects, unless variables are explicitly declared using the `global` primitive. These explicitly named global variables can be handled by treating them as output values. The absence of aliasing eliminates the need to worry about unknown side-effects.

- All the relevant optimizations lend themselves to being described as specializing transformations.

Extending this approach to situations where some of the properties listed above are relaxed is an open problem that would need further investigation.

6.6 Implementation

The compiler has the front-end, the code generator, the type inference engine, and a specialization engine that uses the type information coming out of the type inference engine and specializes the code based on types. The specialization specifications are written in the XML-based language described earlier in this chapter, which is parsed using Apache `Xerces-C`—an XML parsing library for C++ [59]. The entire MATLAB parsing, code-generation, type inference, and specialization infrastructure has been developed from scratch.

While supplying the specialization rules the order of their application can dramatically change the results. For example, specialization rules for scalar operations must be applied *after* type-based specialization has been applied because without the latter all operations are assumed to be generic operations and no scalar operations would be found in the code. In most cases, it is safe to apply type-based specialization rules before any other kind of rules. Some of the rules may also need to be applied repeatedly to get their complete benefit.

Chapter 7

DSP Library

The process of preparing programs for a digital computer is especially attractive, not only because it can be economically and scientifically rewarding, but also because it can be an aesthetic experience much like composing poetry or music.

–Donald E. Knuth

A collection of MATLAB procedures has been identified to serve as a benchmark to test the components of the telescoping library compiler that was developed as a part of this dissertation. Some of these procedures were developed in the DSP group at the Electrical and Computer Engineering Department at Rice University. Others were obtained from the contributed section at MathWorks web-site. This collection of procedures can be seen as a domain-specific library that has been developed by the end-users for their applications.

The table in Figure 7.1 summarizes some of the properties of these MATLAB procedures. The rest of this chapter provides a summary of each procedure.

`jakes_mp1` This procedure computes fast fading signals using the Jakes model. It

name	*description*	*size*		*number of*	
		lines	*bytes*	*inputs*	*outputs*
`jakes_mp1`	Computes fast fading signals with Jakes model	41	1567	4	1
`codesdhd`	Simulates the diversity receiver with Viterbi coding	67	1740	9	6
`newcodesig`	Simulates the transmission and channel of a system with convolutional coding and overlapping symbols	92	2666	9	6
`ser_test_fad`	Computes outage probabilities in a wireless channel	79	2493	7	1
`sML_chan_est`	Computes echoes by a group of cell phones in real-world scenario	303	8729	10	4
`acf`	Computes the auto-correlation of a signal	27	511	2	1
`artificial_queue`	Simulates a queue	18	586	3	1
`ffth`	Computes half-space FFT	49	747	1	1
`fourier_by_jump`	Computes Fourier Transformation	67	1965	3	3
`huffcode`	Computes Huffman codewords	79	2152	2	1

Figure 7.1: Summary of the informal DSP library

is used in an application called `ctss` that simulates a complete system with convolutional coding and overlapping codes. The procedure consists of a single loop that performs trignometric computations on matrices.

`codesdhd` This is a Viterbi decoder that uses other lower level functions. It is the most computationally intensive component of the `ctss` application. Most of its

computation is evenly distributed among the lower level functions.

`newcodesig` Used to simulate the transmitter and the channel of a system within the `ctss` application, it is the second most computationally intensive component. Most of its computation is performed inside a single `for` loop.

`ser_test_fad` This procedure implements a value iteration algorithm for finite horizon and variable power to minimize outage under delay constraints and average power constraints. It is used inside an application that simulates outage minimization for a fading channel. It is invoked inside a doubly-nested loop and itself consists of a five-level deep loop-nest where it spends most of its time.

`sML_chan_est` This is a piece of MATLAB code that implements a block in a SimuLink® system that consists of several interconnected blocks. It primarily consists of a single loop that is inside a conditional statement. The procedure is the most time-consuming part of the entire simulation.

`acf` This procedure to compute auto-correlation of a signal is a part of a collection for time-frequencey analysis. The computation is performed inside a simple `for` loop.

`artifical_queue` Almost all the computation in this small procedure is inside a loop that contains a vector statement that resizes an array. The charateristic feature of this procedure is that it typically operates on huge arrays.

`ffth` This procedure computes an FFT on a `real` vector in half the space and time needed for a general FFT. Essentially, this is a version of FFT specialized for `real` input.

®SimuLink is a registered trademark of MathWorks Inc.

`fourier_by_jump` This procedure implements Fourier analysis by the method of jumps. The implementation has been motivated by the lack of accurate results by the intrinsic MATLAB `fft` function in certain cases. It consists of two loops, only one of which is invoked depending on the value of an input argument.

`huffcode` This procedure computes Huffman codewords based on their lengths. The primary computation inside the procedure occurs in a doubly nested loop. The outer `for` loop encloses a `while` loop that is guarded by an `if` condition.

Chapter 8

Experimental Evaluation

What kind of idea are you?

–Salman Rushdie in *The Satanic Verses*

The first set of experiments was conducted to evaluate the benefits of the novel transformations discovered, along with the relevant optimizations. These experiments were conducted at the source-level to isolate the effects of the transformations. A later section in the chapter describes the experiments conducted to evaluate the library compiler.

8.1 Evaluating Procedure Strength Reduction and Vectorization

Three different DSP applications were studied to evaluate the idea of procedure strength reduction. Procedure vectorization was applicable in one of the cases. All these applications are part of real simulation experiments being done by the wireless

group in the Electrical and Computer Engineering Department at Rice University.

In order to evaluate the idea the applications were transformed by hand carefully applying only the transformations that a practical compiler can be expected to perform and those that are relevant to procedure strength reduction. The transformations included common subexpression elimination, constant propagation, loop distribution, and procedure strength reduction. The transformations were carried out at the source-to-source level and both the original as well as the transformed programs were run under the standard MATLAB interpreter, unless noted otherwise. Applications ranged in size from about 200 to 800 lines and all took several hours to days to complete their runs. Some of the timing results were obtained by curtailing the number of outermost iterations to keep the run times reasonable.

For this study, procedure strength reduction was carried out only at the statement level. It is possible to refine it further and carry it out at the expression level. This would be equivalent to performing an inter-procedural operator strength reduction.

All timing results are from experiments conducted on a 336 MHz SPARC based machine under MATLAB 5.3.

`ctss`

`ctss` is a program that simulates a digital communication system where the data is encoded by a convolutional code and is then transmitted over a wireless channel using a new modulation scheme that exploits the characteristics of the channel to improve performance. The top level program consists of nested loops and procedure calls inside those loops. While these procedures are written in MATLAB and are part of the program, some of these are actually being used as domain-specific library procedures

since they are used by multiple programs.

Figure 8.2(a) shows the improvements resulting from the transformations applied by hand. The first part of the figure shows the performance improvements achieved in various top-level procedures relative to the original running time. Notice that the procedure `jakes_mp1` achieved more than three-fold speedup. These results do not include the dramatic performance improvement that results from vectorizing the loop inside the procedure `jakes_mp1` since that vectorization had already been performed by the user.

The whole program achieved only a little less than 40% speed improvement since most of the time in the application was spent in the procedure `codesdhd` as shown in the second part of Figure 8.2(a). After applying reduction in strength to this procedure its execution time fell from 23000 seconds to 14640 seconds, and accounted for almost the entire performance gain for the whole application.

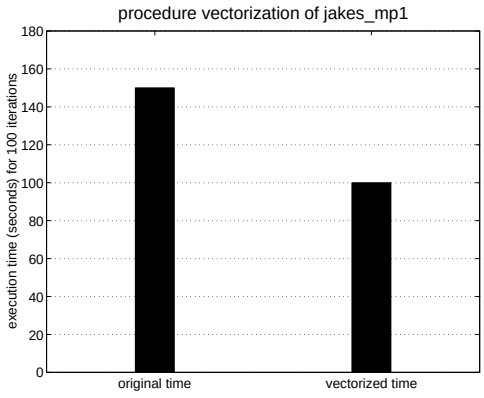


Figure 8.1: Performance improvement in `jakes_mp1`.

In all cases the initialization parts of the procedures were called much less frequently than the iterative parts, effectively making the time spent in the initialization parts comparatively insignificant.

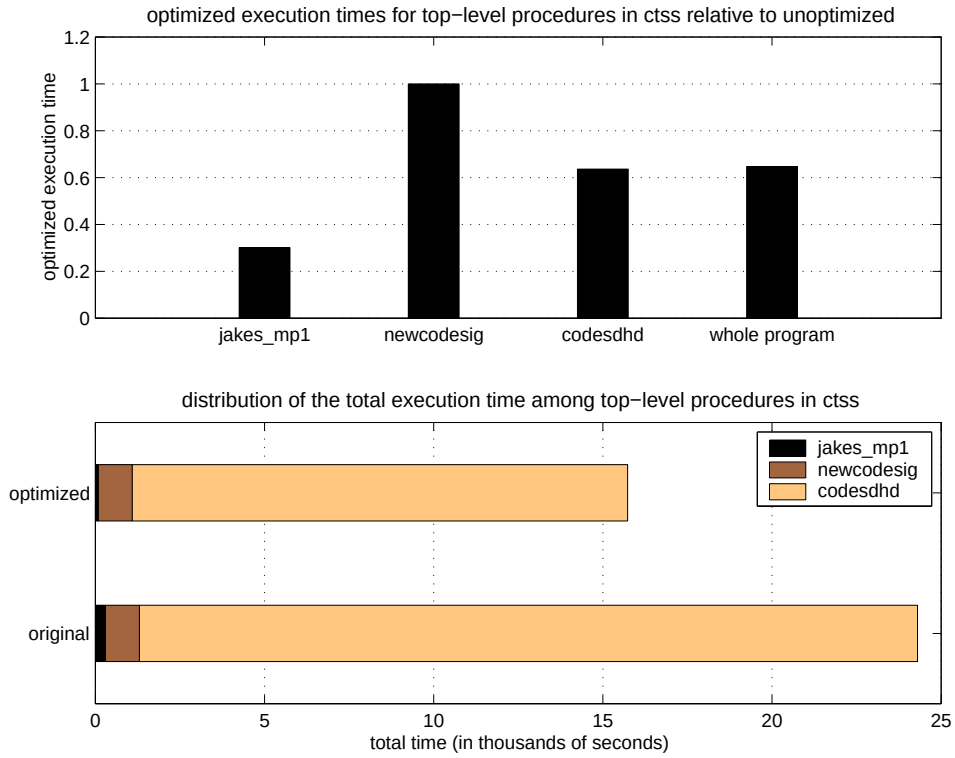
Figure 8.1 shows the timing results for `jakes_mp1` upon applying procedure vectorization. This procedure is fully vectorizable with respect to the loop index of the loop inside which it is called in the main program. The bar graph shows a performance improvement of 33% over the original code for a 100 iteration loop. This performance

gain was almost unchanged down to one iteration loop. Vectorization resulted in a smaller improvement in performance compared to strength reduction because of two possible reasons. First, vectorization of the procedure necessitated an extra copying stage inside the procedure. Second, vectorization resulted in a much higher memory usage giving rise to potential performance bottlenecks due to memory hierarchy. For very large number of loop iterations the effects of memory hierarchy can cause the vectorized loop to perform even poorer than the original code. However, these effects can be mitigated by strip-mining the loop.

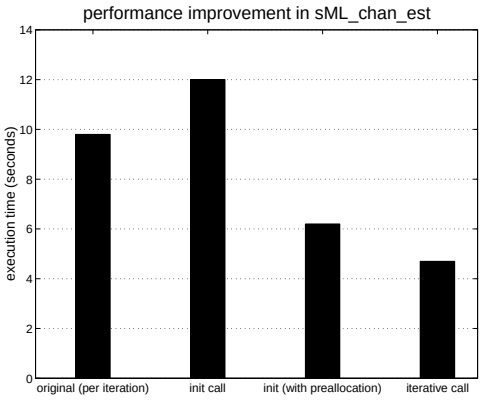
`sML_chan_est`

This application computes the delay, phase, and amplitude of all the echoes produced by the transmissions from a group of cell phones in real world scenario. The application is written under the SimuLink environment provided with MATLAB. One particular procedure was studied in the application, called `sML_chan_est`, which is the one where the application spends most of its time.

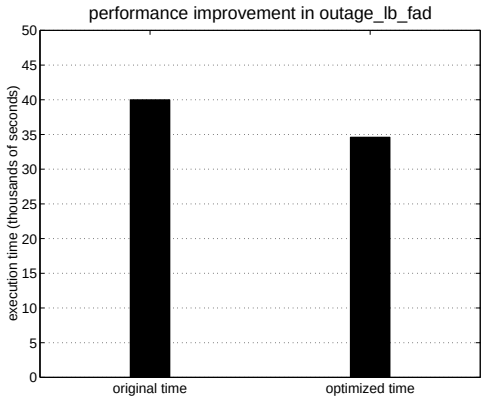
Figure 8.2(b) shows the result of applying strength reduction to this procedure. The graph shows the time taken by the initialization call and the iterative call. The initialization call has a loop that resizes an array in each iteration. If the entire array is preallocated (using `zeros`) then the time spent in initialization drops from 12 seconds (init call) to 6.2 seconds (init with preallocation). This illustrates the value of the techniques needed to handle array reshaping and resizing.



(a) Performance improvements in `ctss`.



(b) Performance improvement in `sML_chan_est`.



(c) Performance improvement in `outage_lb_fad`.

Figure 8.2: Applying procedure strength reduction.

outage_lb_fad

This application computes a lower bound on the probability of outage for a queue transmitting in a time varying wireless channel under average power and delay constraints. It is a relatively small application that spends almost all of its time in a single procedure call. However, complex and deeply nested loops make the application run time very long (several hours).

A straightforward application of procedure strength reduction reduces the runtime of this application by 13.5% (Figure 8.2(c)). This indicates the power of the approach in benefiting even relatively tightly written code.

Effect of Compilation

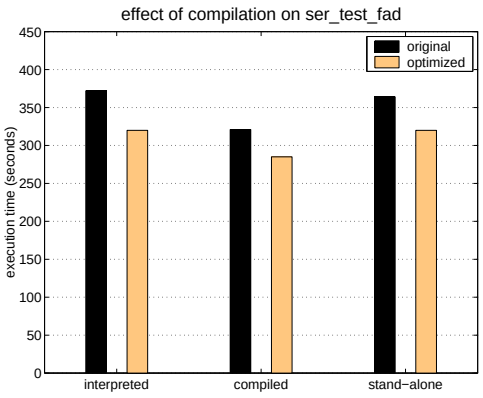


Figure 8.3: Effect of compilation on ser_test_fad.

The mcc compiler by MathWorks works by translating MATLAB functions into C or C++. The compiler simply translates the program into an equivalent sequence of library calls that the interpreter would make in executing the program, without attempting to do any advanced analysis. As a result, the benefit of the compiler can be expected to be maximum when interpretive overheads are high. This would be the case when the application spends considerable time in deeply nested loops.

The application outage_lb_fad was studied for the effects of compilation on procedure strength reduction. The procedure ser_test_fad, which accounts for almost

the entire running time for the application, has a five-level deep loop nest. Figure 8.1 shows the results. While the performance improvement due to procedure strength reduction falls a little for the compiled code, it is still significant at more than 11%. Combining procedure strength reduction and compilation results in a performance improvement of 23.4%.

It should be observed here that this application has very high interpretive overheads due to deep loop nesting and, therefore, represents a best-case scenario for compilation by `mcc`. Other applications showed no significant performance gains due to compilation.

The "stand-alone" column indicates run-times for the code that was compiled to run as a stand-alone application. It is not clear why the stand-alone version of the code is slower than the compiled version that runs under the MATLAB environment. It could be an artifact of dynamically loadable libraries.

8.2 Evaluating the Compiler

Precision of Constraints-based Type Inference

The first set of experiments evaluates the precision of the static constraints-based type inference algorithm. Due to the heavy overloading of operators, MATLAB code is often valid for more than one combination of variable types. For example, a MATLAB function written to perform FFT might be completely valid even if a scalar value is passed as an argument that is expected to be a one-dimensional vector. The results might not be mathematically correct, but the MATLAB operations performed inside the function may make sense individually. As a result, the static type infer-

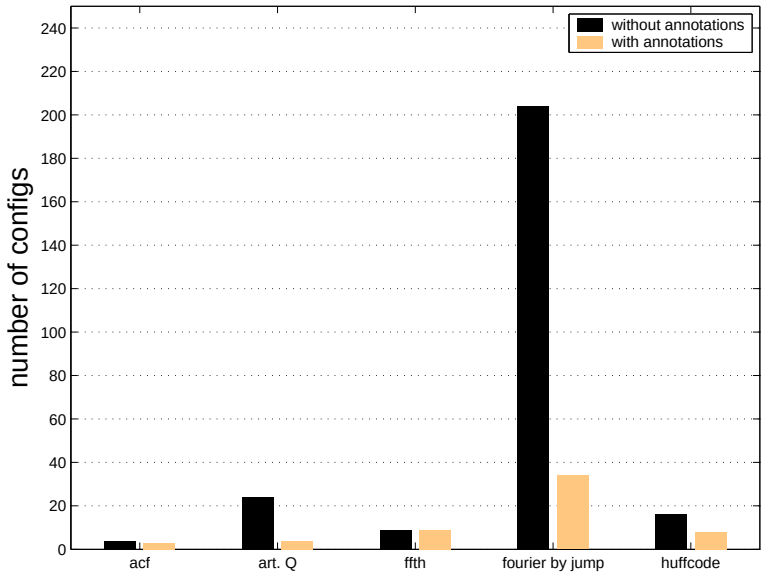


Figure 8.4: Precision of the constraints-based static type inference

ence algorithm can come up with multiple valid type-configurations. Additionally, the limitations enumerated earlier in Section 3.3 can cause the number of configurations to be greater than what would be valid for the given code. This does not affect the correctness since only the generated code corresponding to the right configurations will get used—the extra configurations simply represent wasted compiler effort. In the case of the DSP procedures studied it turns out that if argument types are pinned down through annotations on the argument variables then exactly one type-configuration is valid.

Figure 8.4 shows the number of type-configurations generated for five different DSP procedures by the constraints-based inference algorithm. The left bars indicate the number of configurations generated without any annotations on the arguments. The right bars indicate the number of type-configurations generated when the arguments have been annotated with their precise types that are expected by the library

writer.

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

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

Effectiveness of Slice-hoisting

Having verified that there is a need to plug the hole left by the limitations in the constraints-based inference algorithm, another set of experiments was conducted on the same procedures to evaluate the effectiveness of slice-hoisting. Figure 8.5 shows the percentages of the total number of variables that are inferred by various mechanisms. In each case, exactly one type-configuration is produced, which is the only valid configuration once argument types have been determined through library-writer's annotations. In one case, of `acf`, all the arguments can be inferred without the need for any annotations. The results clearly show that for the evaluated procedures slice-hoisting successfully inferred all the array-sizes that were not handled by the static analysis.

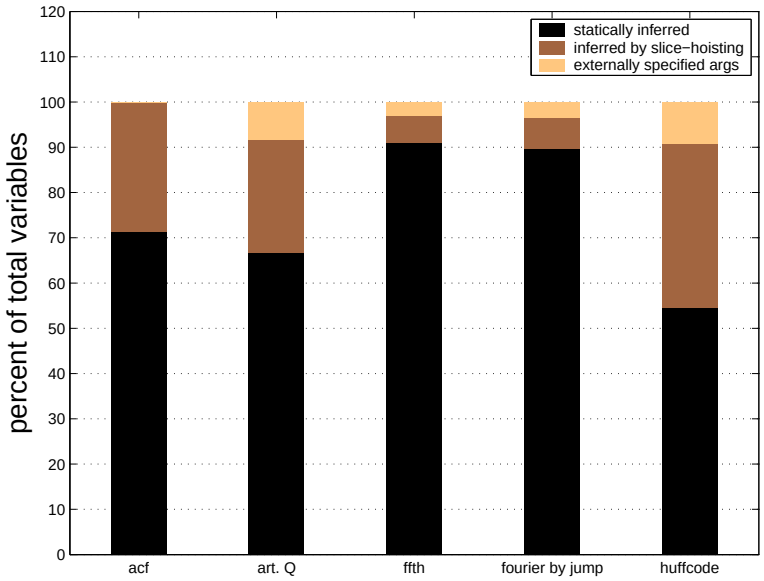


Figure 8.5: Value of slice-hoisting

Type-based Specialization

Figure 8.6 shows the type-based specialization of Jakes. The left two bars are the running times under MATLAB 6.5 and MATLAB 5.3, respectively. There is no bar for MATLAB 5.3 on PowerBook because that version is not available under MacOS X. The reported running times are those of the code running under the MATLAB interpretive environment and not those of the code emitted by `mcc`. Experience in the past has shown that compiling and running the `mcc`-generated code in a stand-alone mode is usually marginally *slower* than running the code under the MATLAB interpreter. The interpreter translates the MATLAB source into an intermediate "byte-code", mitigating the interpretive overheads. It is not clear why the stand-alone version should run slower than that under the interpreter.

The performance is slightly better under MATLAB 6.5 because of the just-in-time compiler that only exists in version 6.5. The rightmost bars are the running times

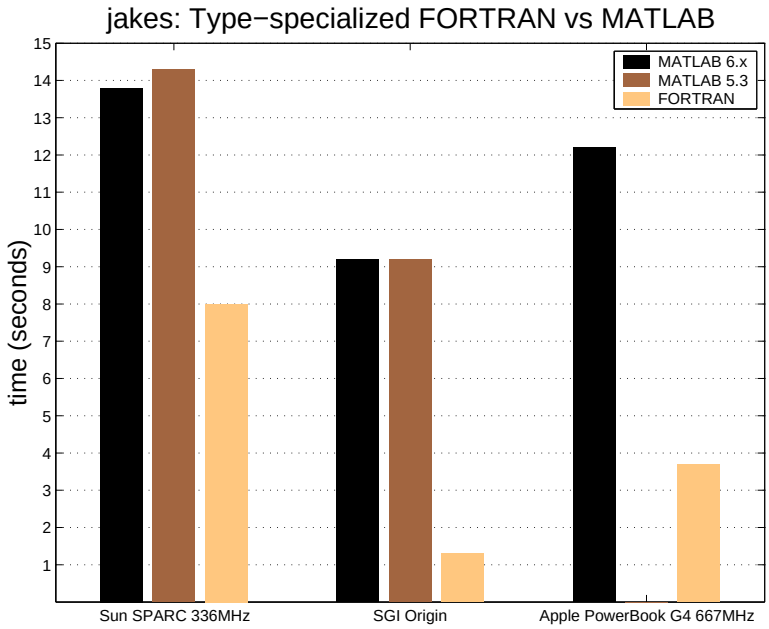


Figure 8.6: Type-based specialization of Jakes

of a Fortran version of the same code that was the result of type-based specialization. No other optimizations were performed on the Fortran version, except those performed by the back-end Fortran compilers. The bars clearly indicate a dramatic performance improvement resulting from this single specialization that far outweighs the performance gains out of just-in-time compilation. Similar improvements have also been seen in the linear algebra domain.

The improvements are greater on SGI than on the other platforms. This could be a result of the more advanced MIPSPro Fortran compiler on SGI that is able to do a better job of optimizing the Fortran code. These performance differences due to the back-end compilers validate the telescoping languages strategy of emitting code in an intermediate language. Since the highly-tuned vendor compilers are often able to do a very good job of compiling programs in lower-level languages, leveraging those

compilers is much more profitable than attempting to generate native machine code directly, which is likely to be of a poorer quality.

Chapter 9

Contributions

> *Perhaps here is the reality we have always suspected. Do your best; you will ultimately slip into history along with the trilobites and other proud personae in this unfolding pageant. If we must eventually fail, what an adventure to be players at all.*
>
> –Stuart Kauffman in *At Home in the Universe*

The biggest contribution of this dissertation is the partial validation of the telescoping languages strategy for library generation in the domain of digital signal processing. The experiments have successfully demonstrated that speculative specialization, driven by library annotations, can result in remarkable improvements in performance of library procedures written in MATLAB.

The telescoping languages strategy was studied for MATLAB libraries for DSP applications. Related work on a linear algebra library, written in MATLAB, indicates that the ideas bear fruit in other domains as well. There are strong indications that similar results can be expected in compiling other numerical high-level languages, such as R+. Libraries used by traditional language systems may also benefit from the specialization techniques described here.

The rest of the chapter describes other specific contributions made during the work on this dissertation.

9.1 Type Inference

A new type inference strategy has been developed in joint work with Cheryl McCosh [13]. The resulting strategy allows type-based specialization of libraries. A study on a linear algebra library has shown that users develop libraries in MATLAB with multiple interpretations of types in mind. Further, type-correlation of arguments is a common occurrence in library procedures. A similar conclusion holds for DSP libraries even though the DSP libraries tend to be written with only one combination of valid types in mind.

Detecting and specializing these library procedures on the intended types can result in significant performance gains over using the most general types for each variable [14]. A dynamic inference strategy for array sizes—called slice-hoisting—was developed to infer those sizes that the static approach fails to handle [12].

9.2 Identification of Relevant Optimizations

A number of well-known optimizations have been identified that result in high payoff in compiling DSP libraries written in MATLAB [10, 11]. These optimizations are expected to yield high performance benefits in other domains as well. The optimizations include loop vectorization, beating and dragging along, common sub-expression elimination, and constant propagation. In addition, utilizing library identities was found to be very valuable for high-level optimizations.

Identifying the high-payoff optimizations is an important step in implementing a telescoping compiler. There is a vast set of optimizations that are known in the compiler community. Since the strategy envisions emitting code in an intermediate language, such as C or Fortran, it pays to focus only on those optimizations that can be applied at the source-level to obtain high performance benefits. Identifying the relevant optimizations guides the process of the development of the telescoping-languages infrastructure in the right direction.

9.3 Novel Inter-procedural Optimizations

Two novel inter-procedural optimizations were discovered—procedure strength reduction and procedure vectorization [10, 11]. Both of these optimizations also fall in the category of high-payoff source-level optimizations.

Procedure strength reduction derives its name from operator strength reduction. It is a slicing technique to perform speculative inter-procedural invariant code motion at library compilation time. Procedure strength reduction can also be combined with operator strength reduction and automatic differentiation, as argued in Chapter 6. Procedure vectorization is a combination of loop-vectorization and loop-embedding.

9.4 A New Approach to Engineering the Compiler

A new approach to engineering a source-level optimizing compiler for high-level scripting languages has emerged in the process of developing a library compiler for MATLAB. This approach has the potential to create remarkable software engineering efficiencies in implementing a compiler.

Traditionally, the optimizing component has been the bulk of a compiler implementation. For the purposes of implementing a telescoping library compiler for MATLAB the relevant optimizations lend themselves to being expressed as specialization transformations in an XML-based language. The language semantics require dependence information. This radically changes the design of the optimizer. The optimizer becomes a specialization engine that uses the dependence information to drive optimizations supplied externally as specializations. This novel architecture of the compiler enables rapid implementation of new optimizations and a much easier verification of the correctness of optimizations.

9.5 Library Generating Compiler

Infrastructure development is a significant effort in compiler research. The work done during the course of this dissertation has resulted in a MATLAB compiler infrastructure consisting of about 25,000 lines of C++ code that uses the Standard Template Library (STL) to succinctly express operations that would otherwise take many more lines of code to implement. The infrastructure includes a MATLAB front-end, a type inference engine, a specialization engine, and a code generator. The Apache open-source Xerces library is used for XML parsing. The infrastructure has been successfully compiled and tested on Solaris, Linux, and MacOS X.

The process of infrastructure development is a long and continuous one. The compiler continues to evolve and is expected to be integrated with the Open64 infrastructure at Rice in order to be released as open source software. Parallel telescoping languages effort for the R language is also expected to benefit from the type inference and the specialization engines.

Chapter 10

Related Work

'The time has come,' the Walrus said,
'To talk of many things:
Of shoes—and ships—and sealing-wax—
Of cabbages—and kings—
And why the sea is boiling hot—
And whether pigs have wings.'
–Lewis Carroll in *Through the Looking-glass*

A consensus is building in the high-performance research community to raise the level of abstraction in programming. As a result, more and more effort is being put into this direction. This chapter reviews some of the work related to various aspects of telescoping languages addressed in this dissertation.

10.1 High-level Programming Systems

APL programming language is the *original* matrix manipulation language. Several techniques developed for APL can be useful in MATLAB compilation, for example, the techniques to handle array re-shaping [1].

In the context of their MaJIC project, work by Menon and Pingali has explored source-level transformations for MATLAB [46, 47]. Their approach is based on using formal specifications for relationships between various operations and trying to use an axiom-based approach to determine an optimal evaluation order. They evaluated their approach for matrix multiplication.

The ROSE project at the Lawrence Livermore National Laboratory aims to develop techniques for high-level transformations of C++ code [50]. Their approach is based on enhancing C++ grammar to a "higher-level grammar" and transforming the C++ abstract syntax tree into that of the higher-level grammar. Multiple grammars can be defined to form a hierarchy. Source-level transformations are carried out by transforming these abstract syntax trees and optimized C++ code is generated by unparsing the final tree. While this project has goals that are similar to those of telescoping languages, the techniques employed and the target language are different. In particular, the ROSE project is targeted at numerical applications written in C++-like object oriented languages.

Automatically Tuned Linear Algebra Software (ATLAS) is a project at the University of Tennessee, Knoxville that is aimed at providing an automatic tuning framework for the LAPACK and the BLAS libraries [58]. The key idea is to separate architecture dependent parts of the code and, while installing the library, use test procedures to find an "optimal" tuning for it. This is similar in spirit to telescoping languages and demonstrates the power of the approach of offline specialization. Since the BLAS constitute the backbone of many numerical computations (including matrix manipulations in DSP), ATLAS complements telescoping languages very nicely—the former focuses on architecture specific low-level optimizations while the latter aims to capture and exploit high-level properties of the source. Another similar project is FFTW,

for computing the Fast Fourier Transform [30].

Many projects have attempted to translate MATLAB into lower-level languages such as C, C++, or Fortran, with varying successes. Some of these target parallel machines using standard message passing libraries. These include the Otter system at the Oregon State University, the CONLAB compiler from the University of Umea in Sweden, and Menhir from Irisa in France [51, 28, 15]. The MATCH project at the Northwestern University attempts to compile MATLAB directly for special purpose hardware [48].

10.2 Type Inference

The MATLAB package comes with a compiler, called `mcc`, by The MathWorks, Inc [44]. The `mcc` compiler works by translating MATLAB into C or C++ and then using a user specified C/C++ compiler to generate object code. The generated code does not indicate that the compiler performs any advanced inter-procedural analysis that is proposed for telescoping languages.

Type inference of variables in high-level languages is well studied in the programming languages community [49]. However, the primary goal of type inference in that community is to prove that a given program behaves "correctly" for all inputs. Numerical quantities are usually treated as one single type. The type inference problem in the context of telescoping languages, in a way, starts where the language theorists stop. Beyond proving the program correct it aims to refine the numerical types to closely match the target language. This adds an engineering dimension to the problem. Specifically, backward propagation and overloaded operators make the problem hard, necessitating approximations to keep the solutions tractable.

Constraint Logic Programming (CLP) has become popular recently [54, 36, 37]. CLP extends the purely syntactic logic programming (typified by linear unification) by adding semantic constraints over specific domains. Using constraints over type domains would fall in the category of Constraint Logic Programming. Some of the well known CLP systems include CHIP [27], CLP($\mathcal{R}$) [38], Prolog-III [17], and ECL^{i}PS^e [57]. While a general purpose CLP system could be employed in solving the constraints within the constraints-based type inference system, a more specialized version is likely to be more efficient under the simplifying assumptions of the type domain.

Type inference for MATLAB was carried out in the FALCON project at the University of Illinois, Urbana-Champaign [26, 25]. The FALCON compiler uses a strategy based on dataflow analysis to infer MATLAB variable types. A simplified version of FALCON's type inference was later used in the MaJIC compiler [7]. Framing the type inference problem as a dataflow problem is inadequate for the purposes of a telescoping compiler, as discussed in Chapter 3. However, the biggest drawback of FALCON's approach to compilation, from the point of view of telescoping languages, is that all function calls are handled through inlining. This leads to a potential blowup in compilation time if library procedures have to be optimized. Separate compilation is unavoidable in the context of telescoping languages—something that is not possible in FALCON. Moreover, recursive functions cannot be handled with this approach and some newer language features of MATLAB have not been addressed.

Building on the FALCON system based on de Rose's work, the University of Illinois and Cornell University have developed a Just In Time (JIT) compiler for MATLAB under their joint project called MaJIC [43]. MaJIC uses a simplified version of FALCON's type inference. However, some recent work enhances the type

inference system with backwardly propagated hints for type speculation [7]. This necessitates iterating over forward and backward passes until convergence. In a general context, such as that encountered in telescoping languages, reasoning about the time complexity—or even termination—of this approach is difficult. In contrast, using propositional expressions to capture the bidirectional flow of information leads to clean graph theoretical algorithms that work for general operators.

Array-size Inference and Slice-hoisting

To perform array-size inference FALCON strategy relies on shadow variables to track array sizes dynamically. In order to minimize the dynamic reallocation overheads it uses a complicated symbolic analysis algorithm to propagate symbolic values of array sizes [56]. Slice-hoisting, on the other hand, can achieve similar goals through a much simpler use-def analysis. Moreover, if an advanced symbolic or dependence analysis is available in the compiler then it can be used to make slice-hoisting more effective. Finally, even very advanced symbolic analysis might not be able to determine sizes that depend on complicated loops while slice-hoisting can handle such cases by converting them to the inspector-executor style.

Conceptually, slice-hoisting is closely related to the idea of inspector-executor style pioneered in the Chaos project at the University of Maryland, College Park by Saltz [52]. That style was used to replicate loops to perform array index calculations for irregular applications in order to improve the performance of the computation loops. In certain cases, hoisted slices can reduce to an inspector-executor style computation of array sizes to avoid the cost of array resizing in loops. However, the idea of slice-hoisting applies in a very different context and is used to handle a much wider

set of situations.

An issue related to inferring array sizes is that of storage management. Joisha and Banerjee have developed a static algorithm, based on the classic register allocation algorithm, to minimize the footprint of a MATLAB application by reusing memory [39]. Reducing an application's footprint can improve performance by making better use of the cache. If a hoisted slice must be executed at runtime to compute the size of an array then the array will be allocated on the heap by Joisha and Banerjee's algorithm. Their algorithm can work independently of—and even complement—slice-hoisting.

Type inference, in general, is a topic that has been researched well in the programming languages community, especially in the context of functional programming languages. Inferring array sizes in weakly typed or untyped languages is undecidable in general, and difficult to solve in practice. Some attempts have been made at inferring array sizes by utilizing dependent types in the language theory community. One such example is eliminating array-bound checking [60].

10.3 Libraries, Specialization, and Partial Evaluation

The Broadway project at the University of Texas, Austin aims at compiling libraries effectively [32, 33]. Many of the project's goals are similar to those of telescoping languages. However, its focus is on libraries without any high-level scripting language, thus avoiding the issue of type inference. While the telescoping languages approach emphasizes specialization aided by library annotations, the Broadway approach relies on a specially designed library annotation language to optimize programs that use

those annotated libraries. The Broadway compiler also attempts to catch library usage errors while the telescoping languages approach makes no attempt at error checking. Instead, it assumes that the script has been debugged under an interpreted environment before the optimizing script compiler is invoked. A scripting language front-end also reduces the possibilities of library usage errors. Finally, the telescoping languages approach has a much greater focus on identifying and discovering high-level optimizations.

The $\mathbf{R}^{\mathrm{n}}$ system developed at Rice in the 1980s contained several elements of managing inter-procedural optimizations that have motivated the telescoping languages approach [19]. A big contribution of the $\mathbf{R}^{\mathrm{n}}$ project was the design of a program compiler that performed a *recompilation analysis* whenever a piece of code was recompiled in order to incrementally update the inter-procedural information of separately compiled components. Thus, the system was able to make use of stored information about pre-compiled procedures to carry out inter-procedural optimization of the whole program and tailer the code of individual procedures to their calling contexts.

Link-time optimization has been proposed to perform inter-procedural optimizations across boundaries of pre-compiled binary libraries or modules [53, 8]. These methods make use of the binary information in the pre-compiled object code at link-time. For compiling scripting languages, higher-level transformations are critical, which are likely to be missed by relying solely on link-time optimizations to perform inter-procedural optimizations. Library annotations and variant generation at the library compilation time enable high-level transformations in telescoping languages.

Cooper, Hall, and Kennedy presented a $O((N+E)N^{k_{max}})$ algorithm to compute procedure clones in order to make more precise compile time information available to a compiler and thus enable more code transformations [18]. Here, N is the number

of procedures in the program, E is the number of call sites, and k_{max} is a constant that is expected to have a small value for most programs. The algorithm proceeds in three phases. The first phase propagates *cloning vectors* down the call graph. Cloning vectors represent cloning opportunities. Second phase merges equivalent cloning vectors using an algorithm that is similar to that of DFA state-minimization. Finally, a third phase performs the actual cloning until the code grows beyond a specified threshold. The algorithm is capable of handling any data flow problem, although, only forward data flow problems, such as constant propagation and alias analysis that can lead to improved opportunities for optimization, are relevant. Some of the techniques developed in this work to manage clones can be reused in code specialization under telescoping languages.

Partial evaluation is a concept that is related to specialization and has been widely studied in the functional programming world [49]. Recently, Elphick, et al., have proposed a scheme for partial evaluation of MATLAB through source-to-source transformation [29]. In order to statically evaluate as much code as possible they make use of a type-system very similar to the one used in this dissertation, however the information is only carried forward, with no backward propagation. Their system performs no specialization even though they admit that that could result in big improvements. From the perspective of the telescoping languages approach, their system does not go far enough. Moreover, the partial-evaluation proposed by Elphick, at al., could be carried out as a pre-pass within the telescoping languages system, thus complementing library compilation.

Multi-stage programming is a systematic method of partial evaluation that has been used to improve the performance of functional languages [55]. It is a powerful technique based on solid theoretical foundations. Unfortunately, it has never been ap-

plied to imperative numerically oriented languages, such as MATLAB. Such languages pose their own challenges that remain unresolved.

10.4 Design of Compilation Systems

Engineering a compiler has been recognized as a difficult problem—sometimes, compared to slaying a dragon [3]. Fortunately, simpler designs can be used for certain situations, e.g., for compiling MATLAB, as demonstrated in Chapter 6.

Source-level transformations were pioneered by David Loveman in his classic paper [42]. Indeed, there are several similarities between the approach outlined in this paper and Loveman's approach of source-to-source transformation. He envisioned performing most optimizations at the source-level and then having a relatively straightforward code generator. The telescoping languages approach relies on highly-tuned vendor compilers for C or Fortran that are the target languages for our MATLAB compiler. Since the output of the compiler is expected to be passed through an optimizing compiler it is possible to ignore those lower-level optimizations that are now standard in most compilers for lower-level languages. The design of the compiler presented in this dissertation is particularly targeted at MATLAB—or MATLAB-like languages—that allow certain simplifications due to their characteristics, such as absence of aliasing. Finally, libraries play a key role in high-level scripting languages. Therefore, most of the optimizations are designed to specialize library procedures and make use of the already optimized lower-level libraries.

Compilers have traditionally followed a phased architecture divided into three major phases: the front-end, the middle optimizer, and the back-end [21]. Most current compiler research focuses on the optimization phase and there exists a large body of

work on designing individual optimizations. An equally important issue in a practical compiler is that of ordering the optimizations. The optimization phase, typically, consists of a series of optimization steps, one feeding into the next. These phases are often not independent of each other but interact with each other closely. Determining the right order of optimizations is a hard problem with a huge space of possibilities, forcing most practical compiler writers to follow rules of thumb based on past experience or intuition. Sometimes, detailed studies can determine that certain phases are best executed together in order to eliminate certain undesirable consequence of their interactions [16]. There have also been attempts to use artificial intelligence search techniques, based on genetic algorithms, to automatically generate right sequences of optimizations [20]. The simplified optimizer for MATLAB enables experimentation with "compiling sequencing" by simply feeding the specialization engine with various permutations of specification sequences. It can also enable combining multiple phases if a specification can be written for the combined transformation.

10.5 Other Related Work

Currying in functional programming languages is a concept related to the idea of procedure strength reduction [22]. Currying abstracts away a function with multiple arguments using a sequence of one-argument functions. The process of currying does not necessarily split the computation across these functions. Procedure strength reduction also serves to reduce the number of arguments to a function, but its entire purpose is to split the computation so that the loop-invariant computation can be hoisted outside the enclosing loop.

Automatic differentiation (AD) is another powerful technique that is related to

procedure strength reduction, as discussed in Section 5.1. It may be possible to apply automatic differentiation to procedure strength reduction, thus leveraging the well established AD techniques.

Chapter 11

Conclusion and Future Work

The known is finite, the unknown infinite; intellectually we stand on an islet in the midst of an illimitable ocean of inexplicability. Our business in every generation is to reclaim a little more land.

–Thomas Henry Huxley

This dissertation work has demonstrated the feasibility of automatic generation of DSP libraries based on the telescoping languages approach, driven by specialization. Experimental studies on applications from the Digital Signal Processing domain support the conclusion. Numerical algebra libraries have also been found to be equally amenable to the approach outlined in this dissertation [14]. This represents a significant step towards the adoption of scripting languages for substantial applications in order to benefit from the immense productivity gains that would inevitably result from making this move.

The type inference solution enables effective type-based library specialization. Specializing on types enables emitting code in the target language using primitive data types of that language, which greatly aids the back-end compiler in generating

good object code. It also allows array variables to be allocated once in the beginning of a procedure call—or even statically—eliminating dynamic resizing of arrays that is a major source of inefficiencies in some library procedures.

Identification of high-payoff optimizations and the discovery of two new optimizations were very important steps towards optimizing the DSP libraries. The study that led to these discoveries also confirmed the value of library writers' annotations in guiding the library compiler to generate variants.

A conceptual framework has been laid out for automatic generation of libraries and implemented as a library compiler for MATLAB based on specialization engine. The unique design of the optimizer, which is driven by specializations specified externally in an XML-based language, enables the representation of all relevant optimizations as specializing transformations. It also facilitates code generation directly in terms of optimized underlying libraries, such as the BLAS.

The type inference engine of the compiler has been tested on an informal DSP library and the effectiveness of type-based specialization has been verified on DSP code. The new optimizations that were discovered have also been experimentally evaluated for their effectiveness on DSP applications.

The work presented here is only the beginning of what promises to be an exciting new approach to compiling for high performance. This chapter provides a glimpse into what is needed in immediate future and conjectures on some of the future directions that have opened up as a result of the findings described so far.

11.1 Short Term Future

There are certain aspects of the system developed during this dissertation that need more work in order for it to be deployable as an open source software. The static type inference system needs to be refined and made more robust in the face of different types of expressions. There also needs to be a control mechanism that would prevent the compiler from getting bogged down by too many cliques. The constraints-based type annotations on procedures may need to be extended to achieve this. One extension that can be easily incorporated is the form of type dependence where the output type of a procedure depends on the *value*, rather than the *type*, of its input arguments. For example, the MATLAB primitive `ones` is such a case.

Variant generation needs to provide safeguards against "variant explosion". For example, if an array is `complex` along one conditional branch and `real` along the other it may be worthwhile generating two different variants corresponding to the two different branches. However, this has the potential of leading to a combinatorial explosion of the number of variants. At some point the alternative of simply inferring the type at runtime and doing a copy becomes more attractive.

There are pending issues regarding supporting code that has a combination of variables whose types have been completely inferred (including those handled through slice-hoisting) and those that must be resolved totally at runtime. MATLAB already provides a mechanism to handle this, but the code generation will need to be adapted to leverage it.

11.2 Future Directions

Apart from the short term polishing work there are many other research directions that have opened up. This section discusses a few.

High-Level Reasoning

As high-level languages raise the level of programming abstraction, and the compiler becomes aware of the high-level operations via the telescoping languages approach, new optimization opportunities open up. This is an issue that was also briefly alluded to earlier in the dissertation. A user script compiler has limited time, but a library compiler could spend a lot of time analyzing various alternatives. A specialization choice made at one point in the program may have a cascading effect on choices available at other points. The compiler will need to operate with a cost-metric in order to evaluate different choices and pick the best possible.

A related issue is that of designing an annotation mechanism that can enable such high-level reasoning. The mechanism will need to be powerful enough to capture specialization as well as describe context dependent transformations along with their cost benefits.

Time-bound Compilation and AI Techniques

User scripts are usually small—the largest of the top-level DSP applications runs no more than a few hundred lines. However, users are willing to wait for a certain amount of time for the compilation to end as long as it does not hinder their interactivity too much. For example, it may be perfectly acceptable for a script compiler to spend two seconds on compilation, instead of two milliseconds. This means that it may

be possible to perform deeper analysis even within the user script compiler rather than settling for a very preliminary peep-hole optimization. However, this analysis will have to be carefully planned to be time bound. It will need to be conducted in phases, generating valid code at the end of each phase, so that the compilation may be terminated whenever a pre-determined time period ends. Such a method of compilation can be highly adaptive, as it will automatically do a better job on a better (faster) machine without sacrificing the user interactivity.

Some lessons from just-in-time compilation may be useful here. However, self-learning AI techniques may be even more useful. The self learning techniques may determine, over time, which combinations of specializations turn out to be useful and which ones are not worth exploring.

Another application of AI techniques in the telescoping languages framework is in learning new contexts for extending the libraries. If an unforeseen context, or a class of closely related contexts, arises sufficiently often in user scripts, the script compiler may be able to provide a feedback to the library compiler in order to specialize the library for that context. The script compiler will need to generalize the context to a class or, at least, recognize a class of closely related contexts and formulate a description for the class. In this way, libraries could be finessed over time and even adapted to specific users.

Dynamic Compilation and the Grid

The idea of computational grid has been attracting a lot of attention recently. One characteristic feature of the grid is its dynamically evolving nature. If the definition of a context is extended to include the run-time environment then the process of

specialization can also account for different run-time scenarios. For example, the libraries could be specialized to specific network topologies, network load signatures, and the characteristics of the available computational nodes.

Library compilation then becomes a continuously evolving process. Some of the ideas outlined in the previous section may also apply here.

Parallelization

One way to write parallel programs is to employ the fork-join model to use parallelized libraries. In such a case parallelization of libraries can be seen as just another form of specialization. The parallelization of libraries could in itself be specialized to a variety of different parallel environments. Further, it may be possible to eliminate the overheads of the fork-join model by, for example, recognizing the redundancy of a join immediately followed by a fork, which would be similar to recognizing library identities.

Several additional, difficult, issues will need to be addressed. Data distribution in a non-uniform memory access environment adds another dimension to the problem of determining identities. The cost model will need to include data communication costs and the compiler will need to be able to estimate communication requirements. Performing redundant computations locally will need to be weighed against fetching data remotely and the selection of the appropriate variants of the libraries will be governed by these tradeoffs.

Automatic Differentiation

Automatic differentiation has been used has a successful technique in numerical applications to automatically generate code for computing the derivative of a function from the given code to compute the function. If this function is invoked inside a loop that computes the value of the function at regular intervals then this technique could be used to perform procedure strength reduction on that function. The value of a function, f, at point $x+\Delta x$ is approximately $f(x)+\Delta x.f^{'}(x)$ if Δx is sufficiently small compared to $f^{''}(x)$, i.e., the rate of change of the derivative. The numerical stability of such an approach will need further investigation.

Diversifying the Domains

Large parts of modern integrated circuits are automatically synthesized and utilize pre-built libraries of components. These components are then stitched together by a synthesizing compiler to build the system. Often the important parts then need to be tuned by hand to maximize performance or certain library components may simply not be usable for the specialized needs of a context. Specialization techniques developed in the telescoping languages approach could automatically optimize the component libraries for specific situations. One of the challenges here would be to develop an annotation language that can be used to describe properties of the circuits and their interfaces.

A very interesting set of possible studies would be to find similar applications in other domains, determine the applicability of "stock" telescoping languages techniques, and develop specific techniques suitable for those domains.

11.3 Final Remark

Compiler technology has come a long way since the earliest compilers that pushed an overwhelming majority of software development from assembly languages to, what were then considered, high-level languages. Today we are poised to make a similar leap from the erstwhile high-level languages to modern domain-specific scripting languages. The prototype compiler built during this dissertation work has established that the technology has advanced enough to make this possible. On the other side of this leap are fantastic productivity gains that millions of users already enjoy for small programs. There is motivation, and there is method; a beginning has been made. It is only a matter of time when we will look back at programming in C and Fortran the way we look back at programming in assembly languages.

Bibliography

[1] Philips S. Abrams. *An APL Machine*. PhD thesis, Stanford Linear Accelerator Center, Stanford University, 1970.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*, chapter Type Checking. Addison-Wesley Pub. Co., 1986.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Pub. Co., 1986.

[4] Frances E. Allen, John Cocke, and Ken Kennedy. Reduction of operator strength. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 79–101. Prentice-Hall, 1981.

[5] John R. Allen and Ken Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

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

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

[8] Andrew Ayers, Richard Schooler, and Robert Gottlieb. Aggressive inlining. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 134–145, 1997.

[9] David Callahan and Ken Kennedy. Analysis of interprocedural side effects in parallel programming environment. *Journal of Parallel and Distributed Computing*, 5:517–550, 1988.

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

[11] Arun Chauhan and Ken Kennedy. Reducing and vectorizing procedures for telescoping languages. *International Journal of Parallel Programming*, 30(4):289–313, August 2002.

[12] Arun Chauhan and Ken Kennedy. Slice-hoisting for array-size inference in MATLAB. In *16th International Workshop on Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science. Springer-Verlag, 2003.

[13] Arun Chauhan, Cheryl McCosh, and Ken Kennedy. Type-based speculative specialization in a telescoping compiler for MATLAB. Technical Report TR03-411, Rice University, Houston, TX, 2003.

[14] Arun Chauhan, Cheryl McCosh, Ken Kennedy, and Richard Hanson. Automatic type-driven library generation for telescoping languages. In *Proceedings of the ACM / IEEE SC Conference on High Performance Networking and Computing*, November 2003.

[15] Stéphane Chauveau and François Bodin. MENHIR: An environment for high performance MATLAB. In *Proceedings of the Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, May 1998.

[16] Clifford Noel Click, Jr. *Combining Analyses, Combining Optimizations*. PhD thesis, Rice University, Houston, TX 77005, February 1995. TR95-252.

[17] Alain Colmerauer. An introduction to Prolog-III. *Communications of the ACM*, 33(7):69–90, July 1990.

[18] Keith D. Cooper, Mary W. Hall, and Ken Kennedy. Procedure cloning. In *Proceedings of IEEE International Conference on Computer Languages*, April 1992.

[19] Keith D. Cooper, Ken Kennedy, and Linda Torczon. The impact of interprocedural analysis and optimization in the $\mathbf{R}^{\mathrm{n}}$ programming environment. *ACM Transactions on Programming Languages and Systems*, 8(4):491–523, October 1986.

[20] Keith D. Cooper, Devika Subramanian, and Linda Torczon. Adaptive optimizing compilers for the 21^{st} century. *The Journal of Supercomputing*, 23(1):7–22, August 2002.

[21] Keith D. Cooper and Linda Torczon. *Engineering A Compiler*. Morgan Kaufmann, December 2003.

[22] Haskell B. Curry and Robert Fayes. *Combinatory Logic*, volume 65 of *Studies in Logic and the Foundation of Mathematics*. North Holland Pub. Co., Amsterdam, 1958.

[23] 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.

[24] Raja Das, Mustafa Uysal, Joel Saltz, and Yuan-Shin Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–479, September 1994.

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

[26] Luiz Antônio DeRose. *Compiler Techniques for Matlab Programs*. PhD thesis, University of Illinois at Urbana-Champaign, 1996.

[27] Mehmet Dincbas, Pascal Van Hentenryck, Helmut Simonis, Abderrahmane Aggoun, Thomas Graf, and François Berthier. The constraint logic programming language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems FGCS-88*, pages 693–702, December 1988.

[28] Peter Drakenberg, Peter Jacobson, and Bo Kågström. A CONLAB compiler for a distributed memory multicomputer. In *SIAM Conference on Parallel Processing for Scientific Computing*, volume 2, pages 814–821, 1993.

[29] Daniel Elphick, Michael Leuschel, and Simon Cox. Partial evaluation of MATLAB. In Frank Pfenning and Yannis Smaragdakis, editors, *Proceedings of the 2nd International ACM Conference on Generative Programming and Component Engineering*, pages 344–363, 2003.

[30] 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, May 1998.

[31] Andreas Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, Philadelphia, PA, 2000.

[32] Samuel Z. Guyer and Calvin Lin. An annotation language for optimizing software libraries. In *Proceedings of the ACM SIGPLAN / USENIX Workshop on Domain Specific Languages*, 1999.

[33] Samuel Zev Guyer. *Incorporating Domain-Specific Information into the Compilation Process*. PhD thesis, University of Texas, Austin, 2003.

[34] Mary Hall, Ken Kennedy, and Kathryn McKinley. Interprocedural transformations for parallel code generation. In *Proceedings of Supercomputing*, pages 424–434, November 1991.

[35] Paul Havlak and Ken Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

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

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

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

[39] Pramod G. Joisha and Prithviraj Banerjee. Static array storage optimization in MATLAB. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2003.

[40] 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.

[41] Kathleen Knobe and Vivek Sarkar. Array SSA form and its use in parallelization. In *25th Proceedings of ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, January 1998.

[42] David B. Loveman. Program improvement by source-to-source transformation. *Journal of the Association of Computing Machinery*, 24(1):121–145, January 1977.

[43] `http://polaris.cs.uiuc.edu/majic/`. MAJIC Project.

[44] `http://www.mathworks.com/`. Mathworks, Inc.

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

[46] Vijay Menon and Keshav Pingali. A case for source level transformations in MATLAB. In *Proceedings of the ACM SIGPLAN / USENIX Workshop on Domain Specific Languages*, 1999.

[47] Vijay Menon and Keshav Pingali. High-level semantic optimization of numerical code. In *Proceedings of ACM-SIGARCH International Conference on Supercomputing*, 1999.

[48] Anshuman Nayak, Malay Haldar, Abhay Kanhere, Pramod Joisha, Nagraj Shenoy, Alok Choudhary, and Prith Banerjee. A library-based compiler to execute MATLAB programs on a heterogeneous platform. In *Proceedings of the Conference on Parallel and Distributed Computing Systems*, August 2000.

[49] ACM SIGPLAN Workshop on Partial Evaluation and Semantics Based Program Manipulation.

[50] Daniel J. Quinlan, Brian Miller, Bobby Philip, and Markus Schordan. Treating a user-defined parallel library as a domain-specific language. In *Proceedings*

of the Workshop on High-Level Parallel Programming Models and Supportive Environments, April 2002.

[51] Michael J. Quinn, Alexey Malishevsky, and Nagajagadeswar Seelam. Otter: Bridging the gap between MATLAB and ScaLAPACK. In *Proceedings of IEEE International Symposium on High Performance Distributed Computing*, August 1998.

[52] Shamik Sharma, Ravi Ponnusamy, Bongki Moon, Yuan-Shin Hwang, Raja Das, and Joel Saltz. Run-time and compile-time support for adaptive irregular problems. In *Proceedings of the ACM / IEEE SC Conference on High Performance Networking and Computing*, November 1994.

[53] Amitabh Srivastava and David W. Wall. Link-time optimization of address calculation on a 64-bit architecture. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 49–60, 1994.

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

[55] Walid Taha. *Multi-stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, November 1999.

[56] Peng Tu and David A. Padua. Gated SSA-based demand-driven symbolic analysis for parallelizing compilers. In *Proceedings of ACM-SIGARCH International Conference on Supercomputing*, pages 414–423, 1995.

[57] Mark Wallace, Stefano Novello, and Joachim Schimpf. *ECL^{i}PS^e: A Platform for Constraint Logic Programming.* William Penney Laboratory, Imperial College, London, 1997.

[58] R. Clint Whaley and Jack J. Dongarra. Automatically Tuned Linear Algebra Software. In *Proceedings of the ACM / IEEE SC Conference on High Performance Networking and Computing*, November 1998.

[59] `http://xml.apache.org/xerces-c/index.html`. Xerces-C: The Apache Xerces XML parser for C++.

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

Appendix A

Procedure Strength Reduction for Nested Loops

Suppose that reduction in strength is applied to a procedure that is called inside a loop nest k levels deep. An execution time is associated with each reduction as shown in Figure A.1.

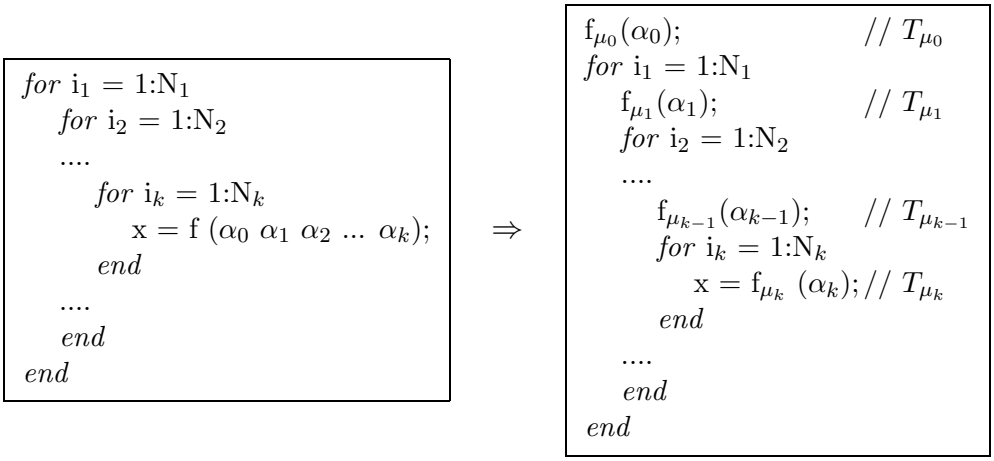


Figure A.1: Procedure strength reduction in a general case.

The original running time, T_θ is given by:

$$T_\theta = \left(\prod_{i=1}^{k} N_i\right) \times T$$

where T is the original running time of one call to `f`. The new execution time, T_ν, for the translated code is given by

$$T_\nu = T_{\mu_0} + (N_1) \times T_{\mu_1} + (N_1.N_2) \times T_{\mu_2} + ... + \left(\prod_{i=1}^{k} N_i\right) \times T_{\mu_k}$$

Thus, the difference in running time, T_Δ, is

$$T_\Delta = T_\theta - T_\nu$$

and relative improvement in speed, T_Δ/T_θ, is given by

$$\frac{T_\Delta}{T_\theta} = 1 - \frac{T_\nu}{T_\theta}$$

or

$$\frac{T_\Delta}{T_\theta} = 1 - \frac{1}{T} \times \left[T_{\mu_k} + \frac{T_{\mu_{k-1}}}{N_k} + \frac{T_{\mu_{k-2}}}{N_{k-1}.N_k} + ... + \frac{T_{\mu_0}}{\prod_{i=1}^{k} N_i}\right]$$

This provides an upper bound on the amount of performance improvement that can be achieved with this method, which is $1 - T_{\mu_k}/T$.

In addition, this equation also provides another useful insight. It is usually the case that the sum of the running times of all f_μs is equal to the original running time T of f, i.e., $T = \sum_{i=0}^{k} T_{\mu_i}$. To obtain maximum performance improvement the summing series in the brackets must be minimized. It is minimized if we can make all $T_{\mu_1}...T_{\mu_{k-1}}$ values zero while maximizing T_{μ_0} given the constraint that all T_{μ_i} sum to T. This corresponds to the intuition that computation should be moved out of the entire loop nest, if possible. However, notice that except the first term, all other terms

in the brackets have iteration range in the *denominator*. Thus, for any reasonably large loop the contribution from all those terms is insignificant. For example, if N_k is 100 the effect of all terms after the first is of the order of only 1%. This leads to the conclusion that except for the case when the innermost loop is very short (in which case the compiler should consider loop unrolling) splitting the procedure `f` more than once may not provide significant benefits. From compiler's perspective, it need not spend much time attempting to reduce procedures multiple times for a multi-level loop since the marginal benefits after the first split are minimal.

Appendix B

XML Schema for Optimization Specification

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema
  targetNamespace="http://www.cs.rice.edu/TeleLangs"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:mint="http://www.cs.rice.edu/TeleLangs"
  elementFormDefault="qualified">

  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      Optimizations as Specializations.
      Rice University, 2003.
    </xsd:documentation>
  </xsd:annotation>

  <xsd:element name="specializations">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="specialization" type="mint:transformation"
                     minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
```

```
</xsd:element>

<xsd:complexType name="transformation">
  <xsd:choice>
    <xsd:sequence>
      <xsd:element name="context" type="mint:preCondition"/>
      <xsd:element name="match" type="mint:stmtList"/>
      <xsd:element name="substitute" type="mint:stmtList"/>
    </xsd:sequence>
    <xsd:sequence>
      <xsd:element name="context" type="mint:preCondition"/>
      <xsd:element name="replaceAllOccurs" type="mint:replacementSpec"/>
    </xsd:sequence>
  </xsd:choice>
</xsd:complexType>

<xsd:complexType name="preCondition">
  <xsd:sequence minOccurs="1" maxOccurs="unbounded">
    <xsd:element name="type">
      <xsd:complexType>
        <xsd:attribute name="var" type="mint:identifier"/>
        <xsd:attribute name="dims" type="xsd:nonNegativeInteger"
                       use="optional"/>
        <xsd:attribute name="intr" type="mint:intrinsic"
                       use="optional"/>
        <xsd:attribute name="constant" type="xsd:boolean"
                       use="optional"/>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="stmtList">
  <xsd:choice minOccurs="0" maxOccurs="unbounded">
    <xsd:element name="simpleStmt" type="mint:simple"/>
    <xsd:element name="loopStmt" type="mint:loop"/>
    <xsd:element name="twoWayBranchStmt" type="mint:twoWayBranch"/>
    <xsd:element name="multiWayBranchStmt" type="mint:multiWayBranch"/>
    <xsd:element name="anyStmt" type="mint:wildCard"/>
  </xsd:choice>
</xsd:complexType>

<xsd:complexType name="replacementSpec">
```

```
  <xsd:sequence>
    <xsd:element name="occurence" type="stmtList"/>
    <xsd:element name="replacement" type="stmtList"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="simple">
  <xsd:sequence>
    <xsd:element name="function" type="identifier"/>
    <xsd:element name="input" type="mint:valList"
                 minOccurs="0" maxOccurs="1"/>
    <xsd:element name="output" type="mint:valList"
                 minOccurs="0" maxOccurs="1"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="loop">
  <xsd:sequence>
    <xsd:element name="condition" type="mint:constOrIdentifier"/>
    <xsd:element name="increment" type="mint:simple"
                 minOccurs="0" maxOccurs="1"/>
    <xsd:element name="body" type="mint:stmtList"
                 minOccurs="0" maxOccurs="1"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="twoWayBranch">
  <xsd:sequence>
    <xsd:element name="condition" type="mint:constOrIdentifier"/>
    <xsd:element name="truebody" type="mint:stmtList"
                 minOccurs="0" maxOccurs="1"/>
    <xsd:element name="falsebody" type="mint:stmtList"
                 minOccurs="0" maxOccurs="1"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="multiWayBranch">
  <xsd:sequence>
    <xsd:element name="condition" type="mint:constOrIdentifier"/>
    <xsd:element name="case">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="value" type="mint:constOrIdentifier"/>
```

```
          <xsd:element name="body" type="mint:stmtList"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="truebody" type="mint:stmtList"
                 minOccurs="0" maxOccurs="1"/>
    <xsd:element name="falsebody" type="mint:stmtList"
                 minOccurs="0" maxOccurs="1"/>
  </xsd:sequence>
 </xsd:complexType>

<xsd:complexType name="wildCard">
  <xsd:attribute name="minCount" type="xsd:nonNegativeInteger"/>
  <xsd:attribute name="maxCount" type="xsd:nonNegativeInteger"/>
</xsd:complexType>

 <xsd:complexType name="valList">
   <xsd:choice minOccurs="1" maxOccurs="unbounded">
     <xsd:element name="val" type="mint:identifier"/>
     <xsd:element name="asection">
       <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="dim" minOccurs="1" maxOccurs="unbounded">
            <xsd:complexType>
              <xsd:choice>
                <xsd:element name="lower" type="mint:constOrIdentifier"/>
                <xsd:element name="upper" type="mint:constOrIdentifier"/>
                <xsd:element name="step" type="mint:constOrIdentifier"/>
              </xsd:choice>
            </xsd:complexType>
          </xsd:element>
        </xsd:sequence>
        <xsd:attribute name="var" type="mint:identifier"/>
        <xsd:attribute name="dims" type="xsd:nonNegativeInteger"/>
       </xsd:complexType>
     </xsd:element>
   </xsd:choice>
 </xsd:complexType>

 <xsd:simpleType name="intrinsic">
   <xsd:restriction base="xsd:string">
     <xsd:pattern value="boolean|integer|real|complex|char"/>
   </xsd:restriction>
```

```
  </xsd:simpleType>

  <xsd:simpleType name="identifier">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="[_a-zA-Z]([a-zA-Z_]|\d)*"/>
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:complexType name="constOrIdentifier">
    <xsd:choice>
      <xsd:element name="const" type="xsd:nonNegativeInteger"/>
      <xsd:element name="symbolic" type="mint:identifier"/>
    </xsd:choice>
  </xsd:complexType>

</xsd:schema>
```

The philosophical lesson learned from this doctorate:

Timeliness is more important than perfection.