Telescoping Languages: A Strategy for Automatic Generation of Scientific Problem-Solving Systems from Annotated Libraries

Ken Kennedy, Bradley Broom, Keith Cooper, Jack Dongarra, Rob Fowler,
Dennis Gannon, Lennart Johnsson, John Mellor-Crummey, and Linda Torczon

July 31, 2000

Abstract

As machines and programs have become more complex, the process of programming applications that can exploit the power of high-performance systems has become more difficult and correspondingly more labor-intensive. This has substantially widened the *software gap*—the discrepancy between the need for new software and the aggregate capacity of the workforce to produce it. This problem has been compounded by the slow growth of programming productivity, especially for high-performance programs, over the past two decades.

One way to bridge this gap is to make it possible for end users to develop programs in high-level domain-specific programming systems. In the past, a major impediment to the acceptance of such systems has been the poor performance of the resulting applications. To address this problem, we are developing a new compiler-based infrastructure, called *MetaScript*, that will make it practical to construct efficient script-based high-level languages from annotated component libraries. These languages are called *telescoping languages*, because they can be nested within one another.

For programs written in telescoping languages, high performance and reasonable compilation times can be achieved by exhaustively analyzing the component libraries in advance to produce a language processor that recognizes and optimizes library operations as primitives in the language. The key to making this strategy practical is to keep compile times low by generating a custom compiler with extensive built-in knowledge of the underlying libraries. The goal is to achieve compile times that are linearly proportional to size of the program presented by the user, rather than to the aggregate size of that program plus the base libraries.

1 Introduction

As high-performance computing platforms continue to increase in complexity, the performance achievable by applications is becoming much more sensitive to program and data organization. Developers of scientific applications thus face an increasingly daunting programming task. This has severely limited the productivity of scientific programmers and inhibited the rate at which high-end computing systems are accepted by the scientific rank and file. Combined with a shortage of application developers sophisticated enough to deal with the architectural complexity, this has led to an effective crisis in high-performance-software development. One way to address this problem is to make it easier for *end users* to solve problems by providing high-level programming systems in which they can rapidly develop new applications using the standard notations of their problem domains.

In point of fact, many high-level problem-solving environments exist today. MATLAB [19] and Mathematica [44] are standard tools for matrix computations and symbolic analysis. Octave [14] is a high-level language originally intended for chemical reactor design. EllPack [24] is a language for describing and solving elliptic partial differential equations. However, most of these do not achieve acceptable performance for complex, computation-intensive applications, especially if they entail substantial programming in the package's scripting language. If we are to make high-level problem-solving systems effective, the performance of applications developed using these systems must be high enough to obviate the need for recoding them in more conventional programming languages such as Fortran 90, C, and C++.

1.1 Current Strategies

High-level problem-solving systems in use today are often constructed from domain-specific libraries and flexible scripting systems. However, because existing scripting languages are typically interpreted and treat libraries as black boxes, the performance of such systems is inadequate for compute-intensive applications. Previous research [13, 7, 9] has shown that this problem can be ameliorated by translating the script to a conventional programming language and then compiling using whole-program analysis and optimization. The problem with this approach is that compilation times are quite long—usually way out of proportion with the length of the user's script.

An alternative strategy, pursued by the POOMA project [1], serves as a model for some of the ideas we will explore in this effort. POOMA is based on a carefully coded C++ library that implements distributed container classes and operations that include overloaded arithmetic operators on these classes. Implementations of overloaded arithmetic operators for container classes in POOMA exploit C++ template instantiation to enable generation of inlined loops that evaluate elementwise expressions on values in containers.

Unfortunately, POOMA suffers from two shortcomings. First, program compilation times are quite long due to the time required for expansion of program representation, and other data structures, used by the template matching and instantiation mechanisms. One large application implemented using POOMA takes roughly an hour to compile on a 24-way parallel machine [20]. Second, POOMA misses opportunities for improving run-time performance because the code that results from instantiating its templates is difficult, if not impossible, to optimize effectively and its operator implementations cannot circumvent this difficulty by directly encoding standard optimizations such as loop fusion and common subexpression elimination.

1.2 The Telescoping Languages Strategy

To overcome these problems, we have devised a new strategy called *telescoping languages*, in which the component libraries, annotated with specifications of high-level, domain-specific optimizations that the compiler could not discover unaided, are extensively analyzed and compiled in advance to synthesize a fast and effective script optimizer.

To make such systems practical, we must overcome the performance limitations without sacrificing architectural portability. This is particularly critical because the primary practical application of this research is intended to be programming systems for large-scale applications on computational "grids"—collections of interconnected heterogeneous computing platforms. Domain-specific problem-solving languages are attractive for this purpose because the language processors can exploit domain knowledge to tailor the implementation to the distributed execution environment.

Preserving portability is a tall order because the usual way to gain performance is by compiling for a specific target machine. In order to avoid implementing a code generator for each new high-performance computing system, we will pursue two approaches for achieving portable performance.

First, we plan to generate *self-tuning* applications by designing and compiling libraries to produce alternative implementations tailored to the properties of each target platform. Where cache-oblivious algorithms [37] are possible, we will use them in the libraries. In other cases, the generated implementations will be extensively tuned by running sample loops on the target machine to find the best values for critical parameters, such as blocking factors for cache locality.

Second, to avoid the need for machine-specific code generators, our translators will produce source code in standard languages that can be fed to each vendor's optimizing compiler. To achieve high performance on each system, the library analysis and preparation step will automatically test the compiler and the code it generates for strengths and weaknesses. It will then generate source code that exploits the strengths and avoids the weaknesses, thus ensuring that the vendor compiler will make the desired performance-enhancing decisions.

Once the library has been analyzed, machine-specific implementations can be selected and integrated in a dynamic compilation pass just prior to execution, as soon as the actual computing environment is known. In addition, library code can reconfigure itself dynamically for better performance once the size and shape of the data are known. This strategy has been adopted by the GrADS project [4, 28], which is building a framework for execution of high-level Grid applications.

The unifying goals of these approaches are to reduce script compilation time and improve portability by investing extra time in a *library analysis and preparation phase*, on the assumption that the domain libraries
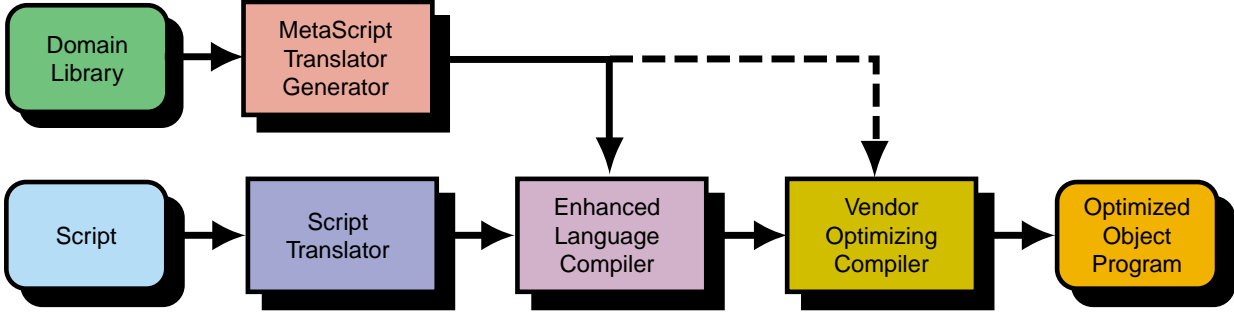
Figure 1: Telescoping Languages

will be recompiled far less often than the scripts that invoke them. By exhaustively exploring the implementation space for library modules in advance, it will be possible to automatically build a powerful framework for transforming and optimizing uses of library primitives. The preliminary analysis and optimization pass may take hours, but it will be worth the cost if the libraries are reused in many different scripts.

2 An Illustrative Scenario

Before describing the underlying technical approaches in detail, we present a scenario to illustrate how our telescoping languages system, which we call *MetaScript*, will be used to build a domain-specific problem-solving system. The process begins when application and library designers decide to build a problem-solving environment based on a library of domain-specific computational procedures. The first approach might be to use a command interpreter that generates calls to individual modules in the library. When this proves too slow for serious applications, the developers decide to try MetaScript. The translation process then follows these steps:

- The team first restructures the application library. The key step is to add annotations specifying program transformations that replace sequences of library calls with equivalent, but more efficient, sequences. In addition, the revised library includes sample calling programs that illustrate the typical usage patterns for the library components. There is also an opportunity to restructure the library so that instead of being composed of large, complex routines hand-optimized for each specific target architecture, it will be constructed out of orthogonal lower-level program units. This takes advantage of the ability of the telescoping compiler to compose such units into efficient modules.

- In a separate step, the designers specify the scripting language to be used and how it is to be translated to the *base language* in which the library is implemented. This specification will be used by a grammar-based translator-generating system. The resulting script translation system will serve as a preprocessing step. Note that this step can be skipped if the designer wishes to use the base language itself as the scripting language. In such cases, MetaScript will build an enhanced optimizing compiler that understands library calls as primitives.

- The enhanced library is processed, perhaps for hours or days, by the MetaScript translator generator. The result, a new translator, can be used to compile a script (that includes library calls) into an extensively optimized program in the base language. This program may include calls to specialized versions of the library routines that use more efficient algorithms, that are tuned to the particular target platform, or that were automatically synthesized to replace a sequence of calls to routines. Because of the machine-specific nature of optimized code, this translation step will need to be redone for each new target machine on which the problem-solving system is to run.

The resulting script compiler will incorporate three phases, illustrated in Figure 1: (1) script translation into a target language program (that may largely consist of library calls), (2) high-level source-to-source optimization of the target language program with the enhanced language compiler generated by Metascript, and (3) compilation of the resulting optimized target language program for the target machine. The entire script compilation process should be limited to a running time that is commensurate with the length of the

script, rather than with the complexity of the routines invoked by that script. In other words, the user should not be surprised by the amount of compilation time used on any particular script. On the other hand, the code produced by the translation process is expected to yield far higher performance because the optimizations will be based on deep knowledge of the library semantics.

3 Intended Applications

To illustrate the leverage offered by the MetaScript system, we discuss several application areas that illustrate some of the key issues.

- MATLAB *compiler*. As mentioned earlier, several groups have developed compilation systems for MAT-LAB programs [13, 9] by hand. MetaScript would automate a large portion of the development of such a compiler.

- *Generalized data distributions*. One of the weaknesses of High Performance Fortran (HPF) was the inflexibility of its built-in data distributions. If compilers could be defined to use a standard distribution library interface, these compilers would be able to incorporate new distributions programmed by users. For example, a library for adaptive distributions based on space-filling (e.g., Hilbert) curves [35] would be easy to add. By extensively pre-compiling such libraries, and by using the annotations to guide the optimization process, the overhead of using such a packaged library might be significantly reduced.

- *POOMA*. The POOMA library provides sophisticated distributed container classes to support scientific calculation. It provides multiple implementations for a variety of container abstractions including distributed arrays, fields, and particles. POOMA uses C++ expression templates [41, 40, 21] to rewrite arithmetic expressions on container objects into inlined loop nests that perform elementwise operations on each of the values in the containers.[1] Since expression templates generate custom inlined code for each use of a container class, compilation can be very time-consuming. Using MetaScript, it will be straightforward to build a library of POOMA-like data structures from which MetaScript can produce an optimized translator that does most of the work of POOMA's expression templates. A MetaScript-generated translator has the potential for generating code that will achieve better runtime performance than C++ expression templates by exploiting knowledge about an operation's semantics and the context in which it is performed. For instance, it could make better use of registers by avoiding the introduction of unnecessary aliases that arise when obliviously instantiating expression templates without regard for the container instances being operated upon. By understanding the semantics of operations on container classes a Metascript-generated translator would also be able to apply program transformation techniques, e.g., loop fusion, that are difficult if not impossible to perform directly on loops that result from expression template instantiation.

- *Grid computing*. Grid computations consist of dynamically assembled, distributed computations that are tied together by an application programmer's control script. For example, NetSolve [8] uses an agent-based design to allow the script code, which can be in Fortran, MATLAB, Mathematica, or Java, to invoke solver library components that are hosted on a set of remote servers. In such a system, the script is the user's "main" program, written in terms of the high-level functionality provided by the remote library components it invokes. The scripts tend to be small programs that are often modified and reused in many different forms. As the real "heavy lifting" in the computation takes place within the remote component, the critical performance factors for the script are that it invoke the best version of required library components for the computation at hand and that it provide an optimal routing of the data between the computationally intensive remote components. An annotated library of remote components could be used to generate a system similar to NetSolve, but with an intrinsic optimization strategy provided from the outset.

4 Telescoping Languages

The goal of the telescoping-languages approach is to automate the construction of high-level problem-solving systems. The process takes as input a set of specially-constructed domain-specific libraries and produces a

---

[1]Recent work explores how to provide similar expression template functionality for Java through the use of partial evaluation [42].

translator for a scripting language into which the library components are integrated as primitive operations. The important subgoals in this process are to achieve high performance and portability of the application, without sacrificing the speed of the script compilation. Based on the assumption library analysis and preparation will occur far less often than script compilation, we are willing to expend significant computation time on library processing to achieve these goals.

To achieve high performance, we plan to use an interprocedural compilation of the script with the domain library components invoked directly or indirectly from the script. The effectiveness of this approach has been established for MATLAB programs by several projects [13, 9]. In addition, we plan to follow the POOMA strategy and leverage specialized optimization strategies coded by the library designer. Unfortunately, experience with these systems indicates that they lead to unacceptably long compilation times (hours in some cases) because the whole program, including all invoked libraries, must be compiled each time a new script is processed.

The *telescoping-language* strategy, depicted in Figure 1, shifts much of the cost of these processes from script compilation time to a library analysis and preparation phase that is done well in advance. The domain-specific, annotated library is provided as input to the MetaScript translator generator, which produces an enhanced base-language compiler, one that understands library entry points and their execution properties as language primitives. In essence, this defines a new problem-solving language consisting of the original base language plus the domain library primitives. In cases where the eventual target machine is known, the library compilation process may precompile parts of the library for the known target, a practice that will be discussed in Section 7, below.

This implementation strategy can be applied iteratively to several different levels of libraries—hence the term "telescoping languages." Once the extended compiler is available, scripts can be translated into calls to the extended library and compiled to efficient object code.

4.1   Key Idea

A key to the success of the telescoping-languages approach is to automatically incorporate knowledge about libraries into an efficient translator that optimizes scripts when they are processed. To do this well, the system must extensively analyze the library and the specification annotations provided about its operations. Based on that analysis, it must distill information about the properties of the operations that the library defines and about the semantic equivalences between sequences of operations. It must then use this information to augment a compilation system with additional analyses and transformations that enable it to optimize sequences of library operations as if they were primitive operations in the underlying language. Investments of substantive processing power in such analysis allows effective high-level optimization of scripts incorporating library calls with only a modest cost at script compilation time.

In many cases, the most promising optimizations will be those specified by the library designer (following POOMA) rather than those found by the compiler. By incorporating algebraic specifications describing transformations allowed by the application domain, especially transformations that the compiler by itself cannot prove are legal, we add to the range of optimizations available to the compiler. In the case of a distributed application, this optimization capability is especially important. If the script program is making a large number of invocations of a library object on a remote host, it may be possible to transform the script to use a single, block-request interface to the library object. This can only happen if the library implements a mutable set of interfaces and this fact is known to the script transformation engine. Specification is a good way to convey this knowledge.

4.2   Component Technologies

Several implementation technologies are critical to the success of this strategy:

- *Library design and specification strategies* help developers organize libraries to facilitate the MetaScript analysis and to specify algebraic properties of library operations (e.g., associativity of matrix multiplication) for use by a compiler. By providing carefully selected sample calling sequences, a library designer can direct reasoning by MetaScript to consider expected operation contexts so that potential optimizations can be identified. Library design is discussed in Section 5, below.

- *Automatic property discovery* eases the burden of specification synthesis faced by library designers, while recognizing properties of the library inputs that are critical to the optimization process. Type

5

and value are examples of properties likely to be important. There are two key aspects to property discovery: identification and propagation. Property *identification* consists of reasoning about which properties of library parameters will be important for optimization. Identification is done by examining library code, specifications, and sample calling sequences. Property *propagation* consists of building into the generated script compiler mechanisms for assessing the impacts on parameter properties of calls to library routines. To do this at library preparation time, MetaScript will construct procedures, called *jump functions*, that provide an efficient mechanism for summarizing the effect of library calls on operand properties such as type and value. Examples of properties that may be critical to some optimizations include the shape of an input operand (matrix rank and size) and whether two operands are effective aliases of one another. Property discovery is discussed in Section 6.1.

- *Strategies for generating efficient, effective script optimizers* are needed so that information about user-defined library routines and their properties can be automatically incorporated into a compiler optimization framework. Such strategies include incorporation of specification-driven transformations on code involving library calls, mechanisms for selecting the most profitable transformations for the context in which the optimization is being performed, and low-level code specialization. These strategies are the subject of Section 6.

5 Library Design and Specification

Although we expect that some benefits will result from the application of the MetaScript translator generator to an arbitrary library, the process is intended to work best if library design uses the following strategies.

- *Organization.* Organization into a small number of computation intensive core components that can be called through a variety of specialized interfaces permits intensive optimization of multiple versions of the core, each specialized to be called from a different context. This organization also facilitates the construction of efficient libraries for a distributed execution environment. An effective organizational principle for maximizing programmer productivity is the separation of orthogonal functionalities. For example, it may be possible to define a set of operations on matrices in one library and a set of data distributions for those matrices in another. Combinations of different matrix operations with different distributions provides a rich collection of operations supported by the library. Orthogonality in design often comes with performance penalties, but the aggressive interprocedural optimizations used by the telescoping compiler will minimize such effects.

- *Adaptivity.* A good high-performance library will be organized to take advantage of special cases that may arise due to properties of the library inputs and the target platform. For example, it may be effective to use one algorithm on large matrices, another on small matrices and vectors. This adaptation to the specifics of the user supplied input and the dynamic nature of the execution environment requires the design of *smart libraries* that will be able to fit the environment and problem specification.

- *Specification of algebraic properties.* A number of simplifying optimizations become possible only with the addition of axioms that specify relationships among the operations defined by the library. For example, if a library provides a stack data structure with push and pop operations, the designer could specify that a push followed by a pop can simply deliver the pushed value. This transformation may eliminate the occurrence of a side effect such as an overflow exception or an extension of the stack representation. These side-effects are non-essential consequences of the implementation chosen for the stack. Without axioms that specifically allow and encourage this transformation, to preserve these side-effects a conventional compiler would not be allowed to perform this transformation even if it were capable of deriving it for the non-overflow case.

- *Specification of contextual transformations.* It can be productive to specify transformations that can be used in certain specific contexts. For example, if a library implements access to an "out-of-core" array or an array blocked for cache, it is useful to replace a single-element fetch within a loop by a block fetch, provided that the loop can be distributed around the fetch operation. To facilitate this transformation, the library designer must specify the algebraic property that the block fetch is equivalent to a loop in which the sole operation is a single-element fetch. In addition, the compiler must discover that a loop containing many operations can be distributed around the fetch, which requires further information

about the library side effects. Such properties are difficult for a compiler to discover but fairly easy for a library designer to specify [32, 43].

- *Sample calling sequences.* Sample calling sequences written by the library and application designers serve to alert the compiler of conditions that will occur in certain idiomatic uses of the library. The compiler is thus driven to optimize for these scenarios. For example, if one of the sample calling sequences passes two arrays that are just shifted aliases of one another to a routine that computes the elementwise average of its inputs, the compiler can synthesize a special-purpose routine for this case that eliminates most of the loads associated with one of the vector operands. It would be implausible to guess that such a usage might be common without such a code sample. By using sample calling sequences as a guide, Metascript could also automatically synthesize composite library operations to replace sequences of library calls when doing so would enable significant optimizations (*e.g.*, when fusing loops from different routines would enhance temporal reuse).

To explore these issues, we are investigating library design strategies that can be effectively used in telescoping languages. The experimental evaluation of these ideas will be pursued through the construction of a number of prototype libraries.

5.1 Evaluating Library Organization Strategies

To support the automatic generation of problem-solving systems on computational grids, we plan to build a collection of scientific software libraries that can serve as testbeds for experimentation with organizational strategies needed for telescoping languages. In addition to generic structuring to work with the MetaScript generator, the libraries will incorporate novel techniques for managing high latencies, low bisection band-widths, and other properties of high-performance computational grids.

The growing gap between the speed of microprocessors and that of memory technology, resulting in deep memory hierarchies, implies that the memory subsystem is a critical performance factor posing increased challenges in the production of software libraries. The research literature contains a growing body of latency tolerant algorithms and program transformations that reorganize code and data to improve latency, bandwidth, and space requirements. Many of these transformations involve multiple library routines, so the libraries will need to be structured so that the right integrated code can be selected by the script compiler based on context

The MetaScript project is exploring strategies for organizing and annotating library routines for use in a telescoping language framework. Beyond providing specifications for axioms describing library entries and their transformations, it will be necessary to systematize methods for organizing code to facilitate discovery by the compiler itself of conditions under which specialized code can be used. One aspect of this will be to isolate computational kernels in regions that can be highly tuned for different target machines, as explained in the section on "portable performance", below.

These efforts will result in a comprehensive toolkit of techniques for designing and constructing high-performance libraries. We also intend to produce suites of prototype libraries that incorporate these techniques, hence providing a basis for a new generation of scientific and engineering problem-solving environments.

5.2 Library Adaptivity and the Design of Smart Libraries

Adaptivity of library functions to the data structures to which they are applied, and their distribution in the case of structures distributed across memory units in a parallel system, are vital for predictable and robust performance. For the Grid, adaptation to different execution environments is also critical for success.

A goal of our research is to explore ways to build adaptive and smart libraries in which the selection among algorithmic and implementation alternatives can be made at the earliest possible moment in the program preparation process. In other words, we would like to use analysis to move the selection to a time prior to execution in order to achieve higher run-time performance.

Adaptivity in libraries takes place at two levels. First, library routines need to adapt their *implementation* to select the most efficient encoding of a given algorithm on the target computing platform. This kind of adaptivity includes instruction selection, loop reordering, and blocking for cache. Second, libraries need to exploit *algorithmic adaptivity*—that is, they must select the best algorithm for the given computing circumstances. An example of the latter is choosing between the conventional $O(N^3)$ matrix multiplication and

Strassen's algorithm, or between Gaussian elimination and some divide-and-conquer scheme for tridiagonal systems of equations. In a parallel execution environment, implementation adaptivity includes blocking and scheduling of communication at each processor as well as global communication strategies, such as how to best realize broadcast or all-to-all communication. Of course, algorithmic adaptivity is critical here as well.

In addition, the library routine should be able to adapt to the data it is given, perhaps changing the algorithm to fit the data and execution environment. Thus, relieving the user of the burden of algorithm and software selection.

In general, algorithmic adaptivity can be achieved through polyalgorithmic library functions, or "overloading", following rules based on performance models built into the library. Selection can be made at compile time, run time, or some combination thereof. For the foreseeable future, algorithmic adaptivity will be the responsibility of the library developer. However, adaptivity for efficiency can be a shared responsibility of library developers, compilers, and development tools. In some cases, the compiler can assist in algorithmic adaptivity, using high-level transformations derived from axioms provided by the library developer

As evidence of the potential impact of adaptivity on the MetaScript system, we point to the Connection Machine Scientific Subroutine Library (CMSSL), developed at Thinking Machines, Inc. In CMSSL, much of the implementation adaptivity was accomplished manually, due to the absence of sufficiently powerful tools. Thus, all code specialization was performed manually or through customized tools for specialized domains, such as WASSEM (Weitek Assembler) for intrinsic functions, as well as the BLAS and the Stencil Compiler for convolution operations. Though this approach resulted in library functions with robust high performance, several single-use tools had to be built, and the strategy severely limited the opportunity for optimizations that crossed the boundaries between library and calling program. In MetaScript, we seek to eliminate these drawbacks through the use of exhaustive library analysis and advance preparation.

5.3   Specifying Algebraic Properties of Library Operations

To support the high-level reasoning needed for our aggressive optimization of scripting languages, a key requirement is that library designers describe precisely the salient algebraic properties of their library components. One promising approach is to express these properties in a formal notation such as a specification language. Specifications in such languages can be formally checked for self-consistency and can be manipulated by automated tools.

Other researchers have explored such axiomatic approaches. Menon and Pingali [33, 32] describe a set of axioms for matrix operations that they embodied in a series of hand-coded routines that are applied in a particular order to optimize MATLAB programs. In contrast, the approach we are developing will automatically derive an optimizer given a set of axioms written in a specification language. Guyer and Lin [18] describe a compiler-based framework and an annotation language to guide optimization of calls to library operations. A library developer annotates library operations with information that indicates which data the operation accesses and modifies, other abstract properties, and hand-coded jump functions to facilitate analysis of data flow at operation call sites. These annotations also include directives indicating when to remove a call to the operation and when to replace it with a call to a more specialized operation. Although their approach shares many of the high-level objectives of our research, it falls short of our goal of using axiomatic equivalences to choose the most efficient equivalent sequences of library calls.

To illustrate our approach, we present some examples of axioms written in the Z specification language [38, 25], which is a typed set theory used extensively for the formal specification of computing systems. Z uses mathematical abstractions to model the data in a system, and uses predicate logic to describe the effect of the operations in the system. It has been applied successfully to significant hardware and software systems, such as the IEEE floating point standard [2].

A Z specification is decomposed into small pieces called *schemas*. Schemas are used to describe both the data in the system and the operations on that data. Z allows specifications to be developed progressively by using the *schema calculus* to combine schemas. The ability to work with incrementally-developed partial specifications is vital for our work with telescoping languages. The schema calculus can also be used to express relations between library operations. Describing the essential algebraic properties of library operations only requires specifying a signature schema for each library component along with properties of individual schema and axioms that relate different compositions of schema. These compositions and axioms are the basis for deriving the desired optimization framework for script compilation.

Semantic equivalence between operations and between compositions of operations are described by stating

8

shape axioms. For example, an axiom that matrix addition is associative

$$\vdash \forall\, m1, m2, m3 : Matrix \bullet$$
$$mmadd(m1, mmadd(m2, m3)) = mmadd(mmadd(m1, m2), m3)$$

would enable the compiler to change the evaluation order of a sequence of those operations.

Although axioms can show equivalence between two different sequences of operations, to use axioms as the basis for optimization requires that the axiom's left- and right-hand-side terms each have associated cost models to help determine when a transformation is profitable. For instance, given axioms for the associativity of matrix multiplication and matrix-vector multiplication:

$$\vdash \forall\, m1, m2 : Matrix;\ v1 : Vector \bullet$$
$$mvmul(mmmul(m1, m2), v1) = mvmul(m1, mvmul(m2, v1))$$

an associated cost model would enable us to recognize the opportunity to replace $(m1 * m2) * v1$, which has $O(n^3)$ complexity, with $m1 * (m2 * v1)$, which has $O(n^2)$ complexity. Similarly, an axiom specifying the associativity of matrix multiplication, for instance, would carry a cost model capturing the benefit of first multiplying the pair of matrices sharing the largest common dimension. This order of operations reduces the overall work necessary. Since array sizes may not be known until run time, the script compilation engine might use the axioms as the basis for generating multi-version code, then put them in a wrapper that selects the version to use at a particular point in a program. At run time, code selection can be accomplished by completing the final steps of the partially evaluated cost models for each of the alternatives.

5.4   Contextual Transformations

It should be possible for the library designer to specify how major computational simplifications can be achieved based on local knowledge of the script. This idea follows the style of the POOMA template library [1] which includes both computational information and library expansion strategies. Guyer and Lin also incorporate specialization transformations [18].

For example, scientific programs often require both the sine and cosine of an angle. Based on a sample calling sequence that shows these two routines invoked in a loop with the same argument, Metascript could automatically synthesize a more efficient composite routine that computes both sine and cosine. The composite routine would achieve higher efficiency by avoiding repetition of case analysis of the angle value and by interleaving the computation of the sine and cosine polynomials to better fill the floating-point execution pipeline.

Returning to the example of the out-of-core array, what we wish to do is to effectively "vectorize" the operand fetch operation for that array. For the compiler system to be able to determine that vectorization is legal, it must be able to discern two facts: (1) that the fetch is not part of any recurrence in the loop, and hence that the loop can be distributed around it, and (2) which library entry implements a "vector fetch." The first of these properties can be determined from hand specifications of dependence properties of the fetch call or by automatic analysis of the library source. The second property must be specified as an algebraic equivalence as described in Section 5.3. In addition, the compiler must have cost models to help determine when transformations like vectorization are profitable. Although it is possible that the dependence patterns could be derived from examining the library code, it is unlikely that the compiler could derive the vectorization relationship between two library entries.

6   Generation of Fast, Effective Optimizers

The goal of optimizer generation is to produce an optimizer that generates excellent code, taking advantage of as many opportunities for transformation as possible while limiting compile time. This means that the compile time must be proportional to the script size rather than to the complexity of the procedures invoked by that script. In other words, we want to avoid full interprocedural analysis and optimization of the script and all the library routines that it calls by precomputing enough information to achieve results comparable to those of a full analysis and optimization.

At the highest level, the script compilation procedure for telescoping languages will be organized into three phases:

- *Analysis of property creation and propagation.* The goal of this phase is to determine at every point in the script a "most precise estimate" of the properties that hold for each parameter of a library invocation.

- *High-level specification-driven transformation.* This phase performs the specification-driven transformations to replace sequences of library calls with more efficient code. The selection of appropriate transformations depends on the types of parameters of the library calls involved, on the program control constructs, and on the data flow patterns that arise from the code sequence. Parameter types after these transformations are recomputed incrementally to support further transformations.

- *Low-level code specialization.* Finally, a code expansion pass substitutes appropriate specialized versions of the library routines at each point of call. The specialized versions are generated via a combination of transformation and selection, based on the properties of the input variables at each point of call. For each invocation, the most specialized code for the given combination of variable properties is selected by a process similar to the *unification* algorithm used in theorem proving [36]. Actual expanded code is loaded from a database of code variants, with a limited transformation step to bound the inlining time. This phase can also select variants based on target machine properties.

Note that through the use of the *noinline* directive in the substituted code, the library designer can preserve procedure invocations where there is no point in performing inlining, the code needs to be kept private, or the invocation is a remote procedure call to a server process on another machine.

Once the script optimizer is finished, the expanded code is presented to the target machine's compiler for the base language to produce machine code.

6.1 Automatic Property Discovery

Property Identification A critical component of the construction of script optimizers will be the determination of special operand properties that can be exploited to achieve high performance. An obviously useful property is the value of a scalar operand, when that operand is a known constant in the calling context. Other examples include type information, the rank and size of matrix operands, and aliasing patterns among operands. In dealing with vector operands, it is useful to know if the elements are contiguous in memory.

Generally, there are three strategies for discovering which properties are critical to optimizing a library:

- Examination of the *library specifications* to discover which properties are essential to carry out some key transformation;

- Examination of the *source code* of the library itself, which may determine that an operand property can yield significant savings. For example, a collection of loops can be interchanged to achieve stride-one access if any dimension as long as the input matrix is contiguous. Discovery of such properties requires that the analyzer reason backward from optimization points to determine parameter properties that permit these optimizations [29]. Although this bears some similarity to *slicing* [23], it actually is more like computation of inverse jump functions [5, 34] or the discovery of weakest preconditions in automated program proof.

- Examination of the *sample calling sequences* provided by the library designer to determine whether the operand examples have useful properties. For example, if a calling sequence passes two matrices that are shifted versions of one another, it may be possible to avoid half the loads in a library routine by saving values in registers between corresponding uses of the two inputs. Even if the loads cannot be avoided, the loops can be restructured to ensure that values are reused while they are still in cache.

Property Propagation Once key properties are identified, the script compiler must be able to determine when these properties hold in the calling program. To do that it must not only recognize their creation, but it also must know when the properties are preserved by other library components. If the script compiler is to do this efficiently, it must be able to determine instantaneously the effect on those properties of other library calls that are on a path from the start of the script to the call being analyzed. *Jump functions* (sometimes called *transfer functions*) provide a way to do this. A jump function for procedure P summarizes the value of an output parameter from P in terms of the values of input parameters. When no precise summary can be computed, the jump function will return a special value indicating that the output parameter is undefined. Where jump functions are well defined, they can make it possible to propagate properties of variables through

a call site without redoing extensive analysis of the called program. Thus, if telescoping languages are to be an effective strategy, the library compilation phase must compute jump functions for every public interface.

Jump functions have been a subject of substantial research on interprocedural analysis [5, 34]. Within the library, they can be constructed by composition if partial jump functions are computed to determine the translation of parameters from a procedure entry to other parameters at call sites within each procedure. If the jump functions become too complicated to compute accurately, they can simply return a gross approximation for the parameter property, effectively saying that it is unknown. A simple jump function can indicate that the value of a reference parameter is unchanged by a library routine [17]. Sophisticated analysis can enable a compiler to compute more complicated jump functions such as equational relationships between outputs and inputs. Havlak's linear congruence relations are an example of this class of relationships [22].

Carefully precomputing jump functions ensures that practically no time is spent during script compilation to determine what properties hold for output parameters from the library. This makes it possible to propagate these properties rapidly throughout the script.

6.2 Specification-driven optimization

A key goal of our research is to devise a program transformation framework that can be instantiated automatically from axiomatic specifications. Axioms indicate relationships between different semantically equivalent sequences of operations. To use an axiom as the justification for a program transformation, the pattern of operations specified by either the left-hand or right-hand side of the axiom must match a pattern of operations in a script.

Using simple text or tree matching to identify replaceable patterns in a script is insufficient for our purposes, because unrelated operations (from an axiom's perspective) may be interleaved in the script. To avoid this problem, Metascript will collect information about operation sequences by traversing a representation of a script's data flow instead. Static-single assignment (SSA) form [12] provides a convenient high-level representation for examining relationships between variable definitions and uses. In some cases, we may need to compute full dependence relations to match axiom specifications.

How best to match operation patterns specified by axioms to a data flow representation for a script that we extract from programs is an open research question. Given a set of axioms, the question is how to identify quickly all of the possible axioms that could be applied and how to select from among them based on associated cost models and the rules for composing cost functions. One approach that we have used successfully in the past is building recognizers based on graph grammars that exhibit the Finite Church-Rosser (FCR) property [15, 30]. However, these methods can converge on local, sub-optimal extrema rather than the global optimum. Another approach is to apply a dynamic programming method to consider sequences of axioms to apply. When an axiom is selected to initiate a restructuring transformation, the translator must consider its effect on the representation to determine if the application of transformation causes additional axioms to match. A possible approach is to shift some of the cost of dynamic programming from script compilation time to library analysis time by precomputing and reasoning about the best axiomatic transformation strategies and costs for expected sequences of operations. When these sequences arise and certain conditions about their context are met, precomputed costs and axiomatic transformations can be incorporated into the dynamic programming calculation at script compilation time without reconsidering all possible alternatives for such sequences.

In addition to using a data flow representation for matching axioms with scripts, another key technology that we will explore as part of this effort is the use of data-flow information to infer the type and shape of dynamically typed operands.

6.3 Low-level Code Expansion

In this phase, the goal is to replace procedure invocations in the script with invocations of more efficient versions of those procedures specialized to the properties of the inputs as estimated at compile time. A subgoal is to do this via a combination of inline expansion and invocation of specialized entries so that the amount of code that must be processed by the target machine's optimizer is not more than linearly proportional to the size of the original script.

Specialization is not a new idea. This strategy has been widely researched in the literature on interprocedural analysis [11, 18] and in work on partial evaluation [27, 3]. Our strategy differs from most of the

previous work in that the specializations in a telescoping language must be generated before the calling program is revealed. Hence, the MetaScript translator generator must prepare specializations for all combinations of parameter properties that lead to significant cost savings. In doing this, it can expend significant computational resources, but it must avoid generating translators that take unacceptably long to process a script.

The MetaScript translator generator will produce specializations in two ways. First, it will use annotations developed by the library designer, as suggested by Guyer and Lin [18]. Second, It will use the critical properties of input parameters, discovered as described in Section 6.1, to specialize a call interface in the library for each set of critical properties that are simultaneously satisfiable. This involves assuming those properties and optimizing the called routine accordingly. As the various specializations are produced, they will be collected in a database that is loaded when the generated translator is initiated. (Of course, the expansions themselves could be pre-computed and stored on disk.) When the script translator encounters a call to a library routine with a collection of known properties for the parameters, it picks the most specialized version of the routine for the estimated parameter properties (or types) by a process known as *unification*. Unification, which originated in theorem-proving literature, requires time linear in the number of parameters of the called routine, assuming the parameter property lattice has no unbounded descending chains [36, 6].

The total number of variants that must be maintained in the compiler's database and the total number of specialized entries in the library can be reduced by generating only a subset of the meaningful variants in the optimizer construction phase. In other words, variants that only differ by small amounts would be combined. Some of the lost performance might be regained by a good optimizer on the vendor end.

Perhaps the best way to illustrate this idea is by an example. Consider the following simple Fortran routine:

```
SUBROUTINE VMP(C, A, B, M, N, S)
   REAL A(N), B(N), C(M), S
   I = 1
   DO J = 1, N
      C(I) = C(I) + A(J)*B(J)
      I = I + S
   ENDDO
END
```

When presented with this routine, the MetaScript analyzer discovers that the increment to the variable `I` is a function of the input parameter `S`. Therefore, the `DO` loop is vectorizable if $S \neq 0$. When this condition is propagated back to the interface and the code is specialized, we get two versions of the code suitable for inlining. For $S \neq 0$, we have:

```
C(1:S*N-S+1:S) = C(1:S*N-S+1:S) + A(1:N)*B(1:N)
```

But for the case of $S = 0$, we get:

```
C(1) = SUM(A(1:N)*B(1:N))
```

These two specialized variants would be saved in a code selection database for the library.

7   Portable Performance

An essential property of a useful script-based application development system is that it be possible to run the applications developed in the system on a variety of target platforms with the expectation of reasonable performance. The problem is that the final target of a script-based application may not become known until script compilation time, long after the libraries that form the basis for the telescoping language are precompiled. In a naive system, this would mean that the entire application, including the library code, would need to be optimized for a specific target platform just prior to execution time. Once again, long compilation times might make this approach impractical.

Telescoping languages provide an opportunity to overcome this problem. Because MetaScript is free to spend a long time compiling libraries, it can generate specialized variants of every implementation in the database for each of a potential collection of target machines. The script compilation process is primarily a high-level transformation phase to be followed by a low-level compilation for the target machine, so the

same high-level version may be effective for all of the potential target systems. In that case, we need only compile one version. Exceptions will occur in cases in which high-level transformations, such as blocking for cache, must be tuned to a specific machine. In those cases, a specialized implementation is tuned during the exhaustive compilation phase for each equivalence class of target machines and the appropriate version is later passed to the low-level compiler. In some cases, such measures can be avoided by using cache-oblivious algorithms [37].

Once preoptimized and compiled versions of each database routine are available, the correct implementation can be selected at link time. In fact, there is no requirement that the same target machine be used for every invocation—in the GrADS project [4], the link phase is replaced by a dynamic optimization phase that distributes work among a heterogeneous collection of processors, tailoring the code and communication to each target machine in the collection.

To summarize, the process of preparing a library for use in a system based on telescoping languages is as follows: (1) Using the compilation process described earlier, compute specialized implementations, along with associated jump functions, for each public interface in the library. These are typically machine-independent. (2) Specialize the code for each potential target machine if necessary, and invoke the target machine's compiler to produce a linkable object module for each target. At script compile time, the specialized implementations are chosen for each library call for the target machine to which that call has been assigned.

The work on optimizing for specific machines builds on previous efforts that use extensive preprocessing to produce near-optimal code for specific machines. As an example, linear algebra is rich in operations which are highly optimizable, in the sense that a highly tuned code may run multiple orders of magnitude faster than a naively coded routine. However, these optimizations are platform specific, such that an optimization for a given computer architecture will actually cause a slow-down on another architecture.

The traditional method of handling this problem has been to produce hand-optimized routines for a given machine. This is a painstaking process, typically requiring many man-months of highly trained (both in linear algebra and computational optimization) personal. The incredible pace of hardware evolution makes this technique untenable in the long run, particularly considering the many software layers (e.g., operating systems, compilers, etc) that also effect these kinds of optimizations, all of which are changing at similar, but independent rates.

A new paradigm has emerged for the production of highly efficient routines—the library incorporates a generator that provides many ways of doing the required operations and uses empirical timings in order to choose the best method for a given architecture. This approach typically uses code generators (i.e., programs that write other programs) to instantiate the different ways of performing a given operation, and has sophisticated search scripts and robust timing mechanisms to find the best ways of performing the operation for a given architecture. This is done just once for a given architecture and reused thereafter.

A number of efforts are using this approach in their design for high performance for example ATLAS at the University of Tennessee [43] and UHFFT at the University of Houston [31]. These efforts use precompiled and optimized kernels that are selected dynamically.

An obstacle to achieving high performance is that the shape and size of many of the application data structures remain unknown until run time. This is particularly true in applications that use adaptive or dynamic data structures, which are increasingly common in sophisticated scientific applications. Within the telescoping languages framework, this problem can be addressed, at the cost of initial code space, by including alternative implementations in the object code for the application and selecting the correct one dynamically at run time in a preprocessing step that takes place right after the parameters in question are known. This strategy is similar to the dynamic compilation strategies in Java [39] and the inspector-executor method for compiling irregular applications for parallel execution [10].

Although these strategies are not unique to telescoping languages, the framework makes it possible to automate many of the steps in tailoring libraries so that the library developer need only identify shared compute-intensive kernels along with test drivers, leaving the specific optimizations to the library precompilation system.

8   Summary

The telescoping languages strategy provides a new approach to the implementation of high-level, domain-specific problem-solving environments. By expending extensive compilation time on preprocessing the

domain-specific libraries that provide functionality to the system, it can be possible to apply a fast compilation strategy to programs written using a scripting language and to generate code that is efficient enough for production use on scientific problems. The fundamental idea is to expend substantive computational resources on a library analysis and preparation phase that builds a translator that recognizes library calls as primitives of the underlying scripting language. For this strategy to be successful, the underlying libraries must be carefully designed and annotated so that high level transformations can be applied and fast specialized code can be selected at script translation time. In addition, the strategy can be used to generate optimized code for different computing platforms, ensuring that good performance is portable across multiple platforms.

The authors have embarked on a collaborative project to build a system called MetaScript for generating efficient script-based high-level programming systems from component libraries developed to support the telescoping language approach. It is our hope that this system will make it easy to produce new high-level languages in which it will be much easier for the end user to develop efficient applications. By so doing it may help ameliorate the shortage of programming talent, particularly in the domain of scientific computing.

References

[1] S. Atlas, S. Banerjee, J. C. Cummings, Paul J. Hinker, M. Srikant, J. V. W. Reynders, and M. Tholburn. POOMA: A high performance distributed simulation environment for scientific applications. In *Proceedings of Supercomputing 95*, San Diego, CA, December 1995.

[2] G. Barrett. Formal methods applied to a floating-point number system. *IEEE Transactions on Software Engineering*, 15(5):611–621, May 1989.

[3] A. Berlin and D. Weise. Compiling scientific code using partial evaluation. *IEEE Computer*, 23(12):25–37, 1990.

[4] Francine Berman, Andrew Chien, Keith Cooper, Jack Dongarra, Ian Foster, Dennis Gannon, Lennart Johnsson, Ken Kennedy, Carl Kesselman, Dan Reed, Linda Torczon, and Rich Wolski. The GrADS Project: Software support for high-level grid application development. Technical Report COMPTR00-355, Rice University, Department of Computer Science, February 2000.

[5] D. Callahan, K. Cooper, K. Kennedy, and L. Torczon. Interprocedural constant propagation. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986.

[6] D. Callahan and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. In *Proceedings of the First International Conference on Supercomputing*. Springer-Verlag, Athens, Greece, June 1987.

[7] Alan Carle, Keith D. Cooper, Robert T. Hood, Ken Kennedy, Linda Torczon, and Scott K. Warren. A practical environment for Fortran programming. *IEEE Computer*, 20(11):75–89, November 1987.

[8] H. Casanov and J. Dongarra. A network-enabled server for solving computational science problems. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(3):212–223, 1997.

[9] S. Chauveau and F. Bodin. Menhir: An environment for high performance Matlab. *Scientific Programming*, 7:303–312, 1999.

[10] A. Choudhary, G. Fox, S. Ranka, S. Hiranandani, K. Kennedy, C. Koelbel, and J. Saltz. Software support for irregular and loosely synchronous problems. *International Journal of Computing Systems in Engineering*, 3(2):43–52, 1993.

[11] K. Cooper, M. W. Hall, and K. Kennedy. A methodology for procedure cloning. *Computer Languages*, 19(2):105–117, February 1993.

[12] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. 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.

[13] Luiz DeRose and David Padua. A MATLAB to Fortran 90 translator and its effectiveness. In *Proceedings of the 10th International Conference on Supercomputing*, May 1996.

[14] J Eaton. Octave. http://www.che.wisc.edu/octave, 1999.

[15] Rodney Farrow, Ken Kennedy, and Linda Zucconi. Graph grammars and global program data flow analysis. In *Proceedings of Seventeenth Annual Symposium on Foundations of Computer Science*, pages 42–56, October 1976.

[16] M. Frigo. A fast Fourier transform compiler. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 169–180, May 1999.

[17] D. Grove and L. Torczon. Interprocedural constant propagation: A study of jump function implementations. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, Albuquerque, June 1993.

[18] Samuel Guyer and Calvin Lin. An annotation language for optimizing software libraries. In *Proceedings of the Second Conference on Domain-Specific Languages*, October 1999.

[19] B. Hahn. *Essential MATLAB for Scientists and Engineers*. Arnold, 1997.

[20] S. Haney. Personal communication on POOMA experience, 1999.

[21] Scott Haney, James Crotinger, Steve Karmesin, and Stephen Smith. PETE, the Portable Expression Template Engine. *Dr. Dobbs Journal*, 24(10), October 1999.

[22] Paul Havlak. *Interprocedural Symbolic Analysis*. PhD thesis, Dept. of Computer Science, Rice University, May 1994. Also available as CRPC-TR94451 from the Center for Research on Parallel Computation and as CS-TR94-228 from the Rice Department of Computer Science.

[23] S. Horowitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, Atlanta, GA, June 1988.

[24] E. N. Houstis and J.R. Rice. The engineering of modern interfaces for PDE solvers. In E. N. Houstis, J. R. Rice, and R. Vichnevetsky, editors, *Symbolic Computation: Applications to Scientific Computing*, pages 89–94. North-Holland, 1992.

[25] J. Jacky. *The Way of Z: Practical Programming with Formal Methods*. Cambridge University Press, 1997.

[26] Lennart Johnsson et al. CMSSL: A scalable scientific software library. In *Proceedings of the Scalable Parallel Libraries Conference*. IEEE Computer Society, 1994.

[27] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Greneration*. Prentice Hall International, New York, NY, 1993.

[28] K. Kennedy. Compilers, Languages, and Libraries. In I. Foster and C. Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*, pages 181–204. Morgan Kaufmann, 1998.

[29] Ken Kennedy. Telescoping languages: A compiler strategy for implementation of high-level domain-specific programming systems. In *Proceedings of International Parallel and Distributed Processing Symposium 2000*, Cancun, Mexico, May 2000.

[30] Ken Kennedy and Linda Zucconi. Applications of a graph grammar for program control flow analysis. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 72–85, January 1977.

[31] Rishad Mahasoom. An adaptive software library for fast Fourier transforms. Master's thesis, Department of Computer Science, University of Houston, Houston, TX, December 1999.

[32] Vijay Menon and Keshav Pingali. A case for source-level transformations in MATLAB. In *Proceedings of the Second Conference on Domain-Specific Languages*, pages 53–65, October 1999.

[33] Vijay Menon and Keshav Pingali. High-level semantic optimization of numerical codes. In *Proceedings of the International Conference on Supercomputing 1999*, pages 434–443, 1999.

[34] Brian Murphy and Monica Lam. Program analysis with partial transfer functions. In *Proceedings of the Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 94–103, January 2000.

[35] M. Parashar, J. C. Browne, C. Edwards, and K. Klimkowski. A common computational infrastructure for adaptive algorithms for PDE solutions. In *Proceedings of Supercomputing '97*, 1997.

[36] M. S. Patterson and M. N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16(2):158–167, 1978.

[37] Harald Prokop. Cache-oblivious algorithms. Master's thesis, MIT Department of Electrical Engineering and Computer Science, June 1999.

[38] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.

[39] Sun Microsystems Inc. Java HOTSPOT performance engine architecture: A white paper about Sun's second generation performance technology, April 1999. http://java.sun.com/products/hotspot/whitepaper.html.

[40] Davd Vandevoorde. valarray¡Troy¿: An implementation of a numerical array. Available at ftp://ftp.cs.rpi.edu/pub/vandevod/Valarray/Documents/valarray.ps, 1995.

[41] Todd L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995. Reprinted in C++ Gems, ed. Stanley Lippman.

[42] Todd L. Veldhuizen. Just when you thought your little language was safe: "expression templates" in Java. Technical Report IUCS 539, Computer Science Department, Indiana University, July 2000.

[43] Clint Whaley and Jack Dongarra. Automatically tuned linear algebra software. In *Proceedings of SC '98*, Orlando, FL, November 1998. IEEE Publications.

[44] S. Wolfram. *The Mathematica Book*. Cambridge Univ. Press, 1999.