Generation of High Performance Domain-Specific Languages from Component Libraries

Ken Kennedy
Rice University
Collaborators

Bradley Broom
Arun Chauhan
Keith Cooper
Jack Dongarra
Rob Fowler
John Garvin

Lennart Johnsson
Chuck Koelbel
Cheryl McCosh
John Mellor-Crummey
Linda Torczon
A Bit of History

- In the beginning, there was machine language (or “assembly” language)

- 1957: Fortran (John Backus, et al., IBM)
  - made it possible for every scientist to develop (high-performance) applications

- Today: programming high end machines is once again the near-exclusive domain of experts.
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
Philosophy

• Compiler Technology = Off-Line Processing

• **Goal**: Abstraction without guilt:
  
  • programming language usability without sacrificing performance

• **Rule**: performance of both compiler and application must be acceptable to the end user
Examples

- PL/I interpretive macro facility
- Fixed macros can be compiled
  - 10x improvement with compilation
- TransMeta “Code Morphing”
- Dynamic compilation of machine code
The Programming Problem

- Programming is complex, particularly on new platforms
- Professional programmers are in short supply (and expensive)
- Programming systems that result in low performance will not be accepted
The Programming Problem: A Strategy

- Make it possible for end users to become application developers:
  - users integrate software components using *scripting languages* (e.g., Matlab, Python, Visual Basic, S+)
  - professional programmers develop software *components*
The Programming Problem: An Obstacle

• Achieving High Performance:
  • translate scripts and components to common intermediate language
  • optimize the resulting program using whole-program compilation
Whole-Program Compilation

Component Library → Translator → Global Optimizing Compiler → Code Generator

Problem: long compilation times, even for short scripts!

Problem: expert knowledge on specialization lost
Telescoping Languages

L1 Component Library

Script

Translator

Compiler Generator

L1 Compiler

Code Generator

Optimized Application

Could run for many hours

understands library calls as primitives

Could run for many hours
Telescoping Languages: Advantages

- Compile times can be reasonable
- High-level optimizations can be included
- User retains substantive control over language performance
  - Generate a new language with user-produced libraries
- Reliability can be improved
Applications
Applications

• Matlab SP (Signal Processing)
  • Optimizations applied to functions

• Statistical Calculations in Science and Medicine in S
  • Hundred-fold performance improvements

• Grid Application Development

• Library Generation and Maintenance

• Component Integration Systems
Procedure Strength Reduction

\[
\text{for } i = 1:N \\
\quad x = x + f(c_1, c_2, i, c_3) \\
\text{end}
\]

\[
\text{for } i = 1:N \\
\quad x = x + f_1(i) \\
\text{end}
\]
Library Generator (LibGen)

- Prof Dan Sorensen (Rice CAAM) maintains ARPACK, a large-scale eigenvalue solver
- He prototypes the algorithms in Matlab, then generates 8 variants in Fortran by hand:
  - ({Double, Complex} x {Symmetric, Nonsymmetric} x {Dense, Sparse})
- Could this hand generation step be avoided?
LibGen Results

- MATLAB
- ARPACK
- LibGen

Comparison of results for sparse symmetric and sparse nonsymmetric cases.
LibGen Results

- dense symmetric
- dense nonsymmetric

Bar chart showing performance comparison of MATLAB, ARPACK, and LibGen for dense symmetric and dense nonsymmetric problems.
Key Technology: Type Inference

- Translation of Matlab to C or Fortran requires type inference
  - At each point in a function, translator must know types as functions of input parameter types ("type jump functions")

- Constraint-based type inference algorithm
  - Polynomial time
  - Can be used to produce different specialized variants based on input types
Value of Specialization

- sparse-symmetric-real
- dense-symmetric-complex
- dense-symmetric-real
- dense-nonsymmetric-complex

0.072 sec
Component Integration

- Component integration systems are important productivity tools.
- Programs constructed from them may be slow because no context-based code improvements can be applied.
- Telescoping languages can be used to construct component integration systems that yield high-performance applications.
Component Integration: A Special Case

- Integration of different component libraries that
  - Implement *data structures* (e.g., sparse matrices)
  - Implement *functions* on data structures (e.g., linear algebra)

- **Hypothesis:** Telescoping languages can make this feasible (with high performance)
Parallelism in Matlab

- Add distributions to Matlab arrays
- Distributions can cross multiple processors
- Use distributions to guide parallelism
- Hide parallelism in library component operations
- Specialize for specific distributions

A(1:100)  A(101:200)  A(201:300)  A(301:400)
Specialization: HPF Legacy

- Library written with template distribution
- Useful distributions specified by library designer (standard + user-defined)
  - **Examples**: multipartitioning, generalized block, adaptive mesh
- Library specialized using HPF compilation technology to exploit parallelism
  - Performed at language generation time
Implementation
Requirements of Script Compilation

• Scripts must generate efficient programs, comparable to those generated from standard interprocedural methods

• Script compile times should be proportional to length of script, not a function of the complexity of the library
Script Compilation

• **Propagate:** move variable property information throughout the program

• **Substitute:** apply high-level transformations

• **Specialize:** select and substitute specialized variants for library calls

  • use the best approximation to the parameter properties for which there is a specialization in the database
Library Analysis and Preparation

- Discover critical properties (types) and construct propagator for them
- Analyze library annotations and construct type-driven macro substitution pass
- Identify and construct specialized variants and selection table
- manage total number of variants
Implementation

- Library + Annotations
- Type Inference
- Type Jump Functions
- Code Generator
- Script
- Type Inference
- Code Generator
- Object Code
- Specialized Variants
- Database
- Return Type Jump Functions
- Database
What Is Done

- Prototype **Database Generator** (Matlab)
- Constructor for type jump functions
- Generation of C code
  - Variant for each input type tuple specified by library developer
  - User-specifed transformations
  - Replacement of calls to Matlab run-time by type-aware invocations of LAPACK
What Is Left

- **Database Generator**
  - Strategies for reducing variants
  - Macro substitution phase
- **Script compiler** (component integration system)
  - Propagate types and select specialized variants
- **Demonstration** on significant component-based applications
Summary

• **Goal:** make professionals, especially scientists and engineers, more productive

• **Challenge:** the need to balance developer productivity against application performance

• **Strategy:** explore technologies that directly translate prototyping languages to production code

http://www.cs.rice.edu/~ken/Presentations/TelescopeHPCS04.pdf