Fortress: A New Programming Language for Scientific Computing

Guy Steele
Sun Fellow

April 28, 2005

The Context of the Research

- Improving programmer productivity for scientific and engineering applications
- Research funded in part by the DARPA IPTO (Defense Advanced Research Projects Agency Information Processing Technology Office) through their program on High Productivity Computing Systems
- Goal is economically viable technologies for both government and industrial applications by the year 2010 and beyond

"To Do for Fortran What Java™ Did for C"

- Catch "stupid mistakes"
  - Make sure array references are not out of bounds
  - Make sure dereferenced pointers are not null
  - Make sure storage is not deallocated prematurely
- Extensive libraries
- Platform independence
- Security model, including type safety
- Dynamic compilation

Sketch of Fortress

- A growable, open language
- Components: management of large projects
- Distributed data and control models
- Type system organized as objects and “traits”
- Advances in syntax

Interesting Language Design Strategy

Wherever possible,
consider whether a proposed language feature
can be provided by a library
rather than having it wired into the compiler.

A Growable, Open Language

- Push decisions out to libraries
- Old model:
  - Study applications
  - Add language features to improve application coding
- Our new model:
  - Study applications
  - Study how a library can improve application coding
  - Add language features to improve *library* coding
- Conjectures:
  - Better leverage, leading to more rapid improvement
  - Enables experimentation with open-source strategies

Making Abstraction Efficient

We assume implementation technology that makes aggressive use of runtime performance measurement and optimization.

- Repeat the success of the Java™ Virtual Machine
- Often faster than static compilers can produce!
- Inlining, loop unrolling, tests for special cases
- Automatic and/or programmer-guided
- IDE that tracks program transformations
- Goal: programmers (especially library writers) need not fear subroutines, functions, and methods for performance reasons

Components: Managing Large Projects

- Management of APIs and code
- Explicitly control linking from within Fortress
- Factoring and parameterization of project components
- Management of unit testing

Data and Control Models

- Data model: shared global address space
- Control model: multithreaded
- Declared distribution of data and threads
  - Integrated into the type system
  - Policies dictated by libraries, not wired in
  - Details of “standard library” may be like HPF or Chapel
- Transactional access to shared variables
  - Atomic blocks
  - Explicit testing and signaling of failure/retry
  - Lock-free (no blocking, no deadlock)

Should Parallelism Be the Default?

- "Loop" is a misleading term
  - A set of executions of a parameterized block of code
  - Whether to order or parallelize those executions should be a separate question
  - Maybe you should have to ask for sequential execution!
- Fortress "loops" are parallel by default

Type System: Objects and Traits

- Traits: like interfaces, but may contain code
  - Based on work by Schärli, Ducasse, Nierstrasz, Black, et al.
- Types, methods, etc., may be parameterized
  - Parameters may be traits or compile-time constants
- Primitive types are first-class
  - Booleans, integers, floats, characters are all objects
- Support for useful mathematical concepts
  - Vectors, matrices, sets, permutations, combinations, associativity, commutativity, complex, intervals, units
  - All *defined* by standard libraries
  - Properties *verified* by automated unit testing

Checking of Physical Dimensions

- Try not to wire a design into the language
  - Use “growing a language” design principles: can this feature be implemented gracefully as a library?
- Express dimensions as generic metaclasses
  - **`Dimension`** is the metaclass of **`Length`** and **`Time`**
  - **`Unit[Length]`** is the metaclass of **`Meter`** or **`Mile`**
- Overload operators to apply to types
  - **`Mass·Length/Time`** is a type
  - Must be the same type as **`Length·Mass/Time`** !
- One special tool for dimensional algebra
  - Metaclass for “free abelian group”

Advances in Syntax

- Support for customary mathematical syntax
  - Libraries define math operators supplied by Unicode
  - Convenient ASCII rendering (like TeX or Wiki notations)
- Syntactic abstraction
  - Support for embedded domain-specific languages
  - Libraries can define subgrammars
  - Subgrammars invoked at specific points in a grammar
  - Precompiled parsers
  - Telescoping languages: beyond libraries to syntax

Conventional Mathematical Notation

- The language of mathematics is centuries old, convenient, and widely taught
- Programming language notation can become closer to mathematical notation
  - Asterisks are for accountants (or conjugate transpose)
  - Letting juxtaposition represent multiplication, as in

    ```
    y = 3 x sin x cos 2 x log log x
    ```

    is challenging but can be done
  - Extra spaces required to allow multicharacter names
  - Unicode allows a wide variety of operators
  - More efficient way to use even plain old ASCII to encode mathematical expressions

Three Forms for Source Code

- Must be able to use **emacs** or **vi** in a pinch
- But a good IDE can support Unicode and even two-dimensional notation

ASCII	UNICODE	Two-dimensional
`rho0 = r DOT r`	$\rho_0 = r \cdot r$	$\rho_0 = r \cdot r$
`v_norm = v / norm v`	$v\_norm = v / \|v\|$	$v_{norm} = \frac{v}{\|v\|}$
`SUM[k=1:n] a[k] x^k`	$\Sigma[k=1:n]\ a[k]\ x^k$	$\sum_{k=1}^{n} a_k x^k$
`C = A UNION B`	$C = A \cup B$	$C = A \cup B$

We Are Actively Studying . . .

- Matlab, Mathematica, Maple, Macsyma
- Various mathematical typesetting languages
- Old languages (Algol, APL, COLASL, MADCAP, MODCAP, the Klerer-May system)
- New languages (Co-Array Fortran, UPC, ZPL, HPF, Fortran 2003, Chapel)

Some Notational Rules We're Exploring

- Brackets after identifiers contain subscripts
  - Even SUM[k=1:n] for **Σ** notation
- Use ^ for any superscript
  - Even A^T for matrix transpose
- Braces { } are for sets, not code blocks
- Brackets [ ] are for arrays and matrices
- Parentheses ( ) are for tuples
  - That includes argument lists as a special case
- Uppercase names (length>1) are operators

Example: NAS Conjugate Gradient (ASCII)

```
conjGrad(A: Matrix[Float], x: Vector[Float]):
        (Vector[Float], Float)
  cgit_max = 25
  z: Vector[Float] = 0
  r: Vector[Float] = x
  p: Vector[Float] = r
  rho: Float = r^T r
  for j <- seq(1:cgit_max) do
    q = A p
    alpha = rho / p^T q
    z := z + alpha p
    r := r - alpha q
    rho0 = rho
    rho := r^T r
    beta = rho / rho0
    p := r + beta p
  end
  (z, ||x - A z||)
```

Matrix[T] and Vector[T] are parameterized interfaces, where T is the type of the elements.

The form x:T=e declares a variable x of type T with initial value e, and that variable may be updated using the assignment operator :=.

Example: NAS Conjugate Gradient (ASCII)

```
conjGrad[Elt extends Number, nat N,
         Mat extends Matrix[Elt,N BY N],
         Vec extends Vector[Elt,N]
        ](A: Mat, x: Vec): (Vec, Elt)
  cgitmax = 25
  z: Vec = 0
  r: Vec = x
  p: Vec = r
  rho: Elt = r^T r
  for j <- seq(1:cgit_max) do
    q = A p
    alpha = rho / p^T q
    z := z + alpha p
    r := r - alpha q
    rho0 = rho
    rho := r^T r
    beta = rho / rho0
    p := r + beta p
  end
  (z, ||x - A z||)
```

Here we make conjGrad a generic procedure. The runtime compiler may produce multiple instantiations of the code for various types E.

The form x=e as a statement declares variable x to have an unchanging value. The type of x is exactly the type of the expression e.

Example: NAS Conjugate Gradient (UNICODE)

```
conjGrad[Elt extends Number, nat N,
         Mat extends Matrix[Elt,N×N],
         Vec extends Vector[Elt,N]
        ](A: Mat, x: Vec): (Vec, Elt)
  cgit_max = 25
  z: Vec = 0
  r: Vec = x
  p: Vec = r
  ρ: E = r^T r
  do j ← seq(1:cgit_max) do
     q = A p
     α = ρ / p^T q
     z := z + α p
     r := r - α q
     ρ₀ = ρ
     ρ := r^T r
     β = ρ / ρ₀
     p := r + β p
  end do
  return (z, ‖x - A z‖)
```

This would be considered entirely equivalent to the previous version. You might think of this as an abbreviated form of the ASCII version, or you might think of the ASCII version as a way to conveniently enter this version on a standard keyboard.

Example: NAS Conjugate Gradient (math)

$$
\begin{array}{l}
conjGrad\,[\,\text{Elt } \textbf{extends } \text{Number},\ \textbf{nat } \text{N},\\
\qquad\quad \text{Mat } \textbf{extends } \text{Matrix}\,[\,\text{Elt}, N\times N\,],\\
\qquad\quad \text{Vec } \textbf{extends } \text{Vector}\,[\,\text{Elt}, \text{N}\,]\\
\qquad]\,(A:\text{Mat},\ x:\text{Vec}):(\text{Vec},\ \text{Elt})\\
\quad cgit_{\max} = 25\\
\quad z:\text{Vec} = 0\\
\quad r:\text{Vec} = x\\
\quad p:\text{Vec} = r\\
\quad \rho:\text{Elt} = r^{\mathrm{T}} r\\
\quad \textbf{for } j \leftarrow \textbf{seq}\,(1:cgit_{\max})\ \textbf{do}\\
\qquad q = A\,p\\
\qquad \alpha = \dfrac{\rho}{p^{\mathrm{T}} q}\\
\qquad z := z + \alpha\,p\\
\qquad r := r - \alpha\,q\\
\qquad \rho_0 = \rho\\
\qquad \rho := r^{\mathrm{T}} r\\
\qquad \beta = \dfrac{\rho}{\rho_0}\\
\qquad p := r + \beta\,p\\
\quad \textbf{end}\\
\quad (z\,,\ \|x - A\,z\|)
\end{array}
$$

It's not new or surprising that code written in a programming language might be displayed in a conventional math-like format. The point of this example is how similar the code is to the math notation: the gap between the two syntaxes is relatively small. We want to see what will happen if a principal goal of a new language design is to minimize this gap.

Comparison: NAS NPB 1 Specification

$z = 0$
$r = x$
$\rho = r^T r$
$p = r$
DO $i = 1,25$
$q = A\,p$
$\alpha = \rho / (p^T q)$
$z = z + \alpha\,p$
$\rho_0 = \rho$
$r = r - \alpha\,q$
$\rho = r^T r$
$\beta = \rho / \rho_0$
$p = r + \beta\,p$
ENDDO
compute residual norm explicitly: $\|r\| = \|x - A\,z\|$

$z : \mathrm{Vec} = 0$
$r : \mathrm{Vec} = x$
$p : \mathrm{Vec} = r$
$\rho : \mathrm{Elt} = r^{\mathrm{T}} r$
for $j \leftarrow$ **seq**$(1 : cgit_{\max})$ **do**
$q = A\,p$
$\alpha = \dfrac{\rho}{p^{\mathrm{T}} q}$
$z := z + \alpha\,p$
$r := r - \alpha\,q$
$\rho_0 = \rho$
$\rho := r^{\mathrm{T}} r$
$\beta = \dfrac{\rho}{\rho_0}$
$p := r + \beta\,p$
end
$(z\,,\ \|x - A\,z\|)$

Comparison: NAS NBP 2.3 Serial Code

```
do j=1,naa+1
   q(j) = 0.0d0
   z(j) = 0.0d0
   r(j) = x(j)
   p(j) = r(j)
   w(j) = 0.0d0
enddo
sum = 0.0d0
do j=1,lastcol-firstcol+1
   sum = sum + r(j)*r(j)
enddo
rho = sum
do cgit = 1,cgitmax
   do j=1,lastrow-firstrow+1
      sum = 0.d0
      do k=rowstr(j),rowstr(j+1)-1
         sum = sum + a(k)*p(colidx(k))
      enddo
      w(j) = sum
   enddo
   do j=1,lastcol-firstcol+1
      q(j) = w(j)
   enddo
   do j=1,lastcol-firstcol+1
      w(j) = 0.0d0
   enddo
   sum = 0.0d0
   do j=1,lastcol-firstcol+1
      sum = sum + p(j)*q(j)
   enddo
   d = sum
   alpha = rho / d
   rho0 = rho
   do j=1,lastcol-firstcol+1
      z(j) = z(j) + alpha*p(j)
      r(j) = r(j) - alpha*q(j)
   enddo
   sum = 0.0d0
   do j=1,lastcol-firstcol+1
      sum = sum + r(j)*r(j)
   enddo
   rho = sum
   beta = rho / rho0
   do j=1,lastcol-firstcol+1
      p(j) = r(j) + beta*p(j)
   enddo
enddo
do j=1,lastrow-firstrow+1
   sum = 0.d0
   do k=rowstr(j),rowstr(j+1)-1
      sum = sum + a(k)*z(colidx(k))
   enddo
   w(j) = sum
enddo
do j=1,lastcol-firstcol+1
   r(j) = w(j)
enddo
sum = 0.0d0
do j=1,lastcol-firstcol+1
   d   = x(j) - r(j)
   sum = sum + d*d
enddo
d = sum
rnorm = sqrt( d )
```

Our Key Design Themes

- Make abstraction efficient
  - Aggressive static and dynamic optimization
- Make stupid mistakes impossible
  - And make clever mistakes relatively unlikely
- Design the language to be grown by users
  - Rich library language ⇒ simple application languages
- Make parallelism normal and tractable
  - Identify and support standard communication patterns
- Emulate standard mathematical notation
  - Reduce translation effort from science to computation