Fortress: A New Programming Language for Scientific Computing

Sukyoung Ryu

Joint work with Eric Allen, David Chase, Christine Flood, Joseph Hallett,
Victor Luchangco, Jan-Willem Maessen, Guy L. Steele Jr., and Sam Tobin-Hochstadt

Sun Microsystems Laboratories

March 20, 2007

Outline

- Fortress Programming Language
  - > Growing a Language
  - > Mathematical Notation
  - > Parallelism by Default
- Formalism in Fortress
- Project Fortress

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 High Productivity Computing Systems program
- Goal is economically viable technologies for both government and industrial applications by the year 2010 and beyond

The Background of the Research

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

- Catch "stupid mistakes" (like array bounds errors)
- Extensive libraries (e.g., for nework environment)
- Security model (including type safety)
- Dynamic compilation
- Platform independence
- Multithreading

- Make programmers more productive

Key Ideas

- Don't build the language—grow it
- Make programming notation closer to math
- Ease use of parallelism

Outline

- Fortress Programming Language
  - Growing a Language
  - Mathematical Notation
  - Parallelism by Default
- Formalism in Fortress
- Project Fortress

Growing a Language

- Languages have gotten much bigger.
- You can’t build one all at once.
- Therefore it must grow over time.
- What happens if you design it to grow?
- How does the need to grow affect the design?
- Need to grow a user community, too.

See Steele, “Growing a Language” keynote talk, OOPSLA 1998; *Higher-Order and Symbolic Computation* **12**, 221–236 (1999)

What Primitive Data Types to Include?

- Integers and floating-point (what sizes? bignums?)
- Complex numbers, rational numbers, intervals
- Arrays, vectors, and matrices
- Rational intervals, complex intervals
- Complex vectors and matrices
- What about physical units (meters, kilograms)?

“I might say ‘yes’ to *each* one of these, but it is clear that I *must* say ‘no’ to *all of them*!”

Interesting Language Design Strategy

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

Types Defined by Libraries

- Lists, vectors, sets, multisets, and maps
  - Like C Standard Template Library, but better notation

$$\langle 1,2,4,3,4 \rangle \qquad A \cup \{1,2,3,4\}$$

$$[3\ 4\ 5] \times [1\ 0\ 0]$$

- Matrices and multidimensional arrays
- Integers, floats, rationals, with physical units

$$m\colon \mathbb{R}\ \text{Mass} = 3.7\ \text{kg}$$
$$\mathbf{v}\colon \mathbb{R}^3\ \text{Velocity} = [3.5\ 0\ 1]\ \text{m/s}$$
$$\mathbf{p}\colon \mathbb{R}^3\ \text{Momentum} = m\ \mathbf{v}$$

ASCII ("Wiki-like markup") Notation

- Lists, vectors, sets, multisets, and maps
  - > Like C Standard Template Library, but better notation

```
<|1,2,3,4|>        A UNION {1,2,3,4}
    [3 4 5] CROSS [1 0 0]
```

- Matrices and multidimensional arrays
- Integers, floats, rationals, with physical units

```
m: RR Mass = 3.7 kg_
_v: RR^3 Velocity = [3.5 0 1]
 m_/s_
_p: RR^3 Momentum = m _v
```

- Data structures may be local or distributed

Sample Code: Algebraic Constraints

```
trait BinaryPredicate⟦T extends BinaryPredicate⟦T, ∼⟧, opr ∼⟧
   opr ∼(self, other: T): Boolean
end

trait Symmetric⟦T extends Symmetric⟦T, ∼⟧, opr ∼⟧
      extends { BinaryPredicate⟦T, ∼⟧ }
   property ∀(a: T, b: T) (a ∼ b) ↔ (b ∼ a)
end

trait EquivalenceRelation⟦T extends EquivalenceRelation⟦T, ∼⟧, opr ∼⟧
      extends { Reflexive⟦T, ∼⟧, Symmetric⟦T, ∼⟧, Transitive⟦T, ∼⟧ }
end

trait Integer extends { CommutativeRing⟦Integer, +, −, ·, zero, one⟧,
                        TotalOrderOperators⟦Integer, <, ≤, ≥, >, CMP⟧,
                        . . . }
   . . .
end
```

(This is actual Fortress library code.)

Our Vision

With key algorithms in libraries (cf. MATLAB), application code can be concise, therefore easier to check against design specifications.

Outline

- Fortress Programming Language
  - Growing a Language
  - Mathematical Notation
  - Parallelism by Default
- Formalism in Fortress
- Project Fortress

Conventional Mathematical Notation

- The language of mathematics is centuries old, concise, convenient, and widely taught.
- Parsing mathematical notation is a challenge
  - Subtle reliance on whitespace: $\{ |x| \mid x \leftarrow S, 3 \mid x \}$
  - Semantic conventions: $y = 3\ x \sin x \cos 2\ x \log \log x$

What Syntax is Actually Wired In?

- Parentheses ( ) for grouping
- Comma , to separate expressions in tuples
- Semicolon ; to separate statements on a line
- Dot . for field and method selection
- Conservative, traditional rules of precedence
  - A dag, not always transitive (examples: **A+B>C** is okay; so is **B>C** ∨ **D>E**; but **A+B** ∨ **C** needs parentheses)

Libraries Define . . .

- Which operators have infix, prefix, postfix definitions, and what types they apply to

```
opr -(m:ℤ,n:ℤ) = m.subtract(n)
opr -(m:ℤ) = m.negate()
opr (n:ℕ)! = if n=0 then 1 else n·(n-1)! end
```

- Whether a juxtaposition is meaningful

```
opr juxtaposition(m:ℤ,n:ℤ) = m.times(n)
```

- What bracketing operators actually mean

```
opr ⌈x:ℝ⌉ = ceiling(x)
opr |x:ℝ| = if x<0 then -x else x end
opr |s:Set| = s.size
```

Simple Example: NAS CG Kernel (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||)


(z,norm) = conjGrad(A,x)
```

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

Simple Example: NAS CG Kernel (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)
  cgit_max = 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||)


(z,norm) = conjGrad(A,x)
```

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

Simple Example: NAS CG Kernel (Unicode)

```
conjGrad⟦Elt extends Number, nat N,
         Mat extends Matrix⟦Elt,N×N⟧,
         Vec extends Vector
        ⟧(A: Mat, x: Vec): (Vec, Elt)
  cgit_max = 25
  z: Vec := 0
  r: Vec := x
  p: Vec := r
  ρ: Elt := r^T r
  for 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
  (z, ‖x - A z‖)
```

This would be considered entirely equivalent to the previous version. You might think of this as an abbre-viated 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.

Simple Example: NAS CG Kernel

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

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 $i = 1, 25$
$\quad q = A\,p$
$\quad \alpha = \rho / (p^T q)$
$\quad z = z + \alpha\,p$
$\quad \rho_0 = \rho$
$\quad r = r - \alpha\,q$
$\quad \rho = r^T r$
$\quad \beta = \rho / \rho_0$
$\quad p = r + \beta\,p$
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** $(1 : cgit_{\mathrm{max}})$ **do**
$\quad q = A\,p$
$\quad \alpha = \dfrac{\rho}{p^{\mathrm{T}} q}$
$\quad z := z + \alpha\,p$
$\quad r := r - \alpha\,q$
$\quad \rho_0 = \rho$
$\quad \rho := r^{\mathrm{T}} r$
$\quad \beta = \dfrac{\rho}{\rho_0}$
$\quad p := r + \beta\,p$
end
$(z,\ \|x - A\,z\|)$

Comparison: NAS NPB 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 )
```

Outline

- Fortress Programming Language
  - Growing a Language
  - Mathematical Notation
  - Parallelism by Default
- Formalism in Fortress
- Project Fortress

Parallelism Is Not a Feature!

- Parallel programming is not a goal, but a pragmatic compromise.
- It would be a lot easier to program a single processor chip running at 1 PHz than a million processors running at 20 GHz.
  - We don't know how to build a 1 PHz processor.
  - Even if we did, someone would still want to strap a bunch of them together!
- Parallel programming is difficult and error-prone.

Questions

Can we encapsulate parallelism in libraries?

Will this separation be effective?

Should Parallelism Be the Default?

- "Loop" can be 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
- Fortress "loops" are parallel by default
  - > This is actually a library convention about generators
  - > You get sequential execution by asking for it specifically

In Fortress, Parallelism Is the Default

```
for i←1:m, j←1:n do
  a[i,j] := b[i] c[j]
end
```

`1:n` is a generator

```
for i←seq(1:m) do
  for j←seq(1:n) do
    print a[i,j]
  end
end
```

`seq(1:n)` is a sequential generator

```
for i←1:m, j←i:n do
  a[i,j] := b[i] c[j]
end
```

`a.indices` is a generator for the indices of the array `a`

`a.indices.rowMajor` is a sequential generator of indices

```
for (i,j)←a.indices do a[i,j] := b[i] c[j] end
for (i,j)←a.indices.rowMajor do print a[i,j] end
```

- Generators (defined by libraries) manage parallelism and the assignment of threads to processors

Loops, Reducers, Comprehensions

$$\mathbf{for}\ k \leftarrow 1:n\ \mathbf{do}\ \ print\ k\ \ \mathbf{end}$$

$$y = \sum_{k \leftarrow 1:n} a_k x^k$$

$$w = \sum S \qquad \text{(* same as } \sum_{x \leftarrow S} x \text{ *)}$$

$$v = \bigcap_{\substack{k \leftarrow S \\ prime\ k}} arrayOfSets_k$$

$$z = \mathrm{MAX}_{(j,k) \leftarrow a.indices} \left| a_{j,k} - b_{j,k} \right|$$

$$B = \{ f(x, y) \mid x \leftarrow S, y \leftarrow A, x \neq y \}$$

$$l_{\text{triangle}} = \left\langle \frac{x(x+1)}{2} \,\middle|\, x \leftarrow 1:100 \right\rangle$$

Loops, Reducers, Comprehensions

```
for k←1:n do print i end

y = ∑[k←1:n] a[k] x^k
w = ∑S                    (* same as ∑[x←S] x *)
v = ⋂[k←S, prime k] arrayOfSets[k]
z = MAX[(j,k)←a.indices] |a[j,k]-b[j,k]|

B = { f(x,y) | x←S, y←A, x≠y }
l_triangle = ⟨ x(x+1)/2 | x←1:100 ⟩
```

Parallelism in Fortress

- Regions describe machine resources like CPU and memory and their properties.
- Distributions describe how to map aggregates onto regions.
- When a data structure (or its index set) is used as a generator, the parallelism of the generator reflects the distribution of the data structure.

Our Key Design Themes

- Make stupid mistakes impossible
  - And make clever mistakes relatively unlikely
- Design the language to be grown by (expert) users
  - Rich library language enables simple application languages
- Make abstraction efficient
  - Aggressive static and dynamic optimization
- Make parallelism tractable
  - Appropriate abstractions for managing thread and data distribution
- Emulate standard mathematical notation
  - Reduce the effort of translating from science to computation

Outline

- Fortress Programming Language
- Formalism in Fortress
- Project Fortress

Formalism for the Fortress Programming Language

Eric Allen
Eric.Allen@sun.com

Sukyoung Ryu
Sukyoung.Ryu@sun.com

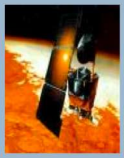
Joe Hallett
Joseph.Hallett@sun.com

The Value of Formal Methods

Ariane 5

A data conversion from 64-bit floating point to 16-bit signed integer value raised an uncaught Overflow exception.

Result: The launcher was destroyed 40 seconds into the flight. The launch cost of an Ariane 5 was $180 million.



Mars Climate Orbiter

Orbiter software represented Force Time in Ns. Ground software represented Force Time in lbf s.

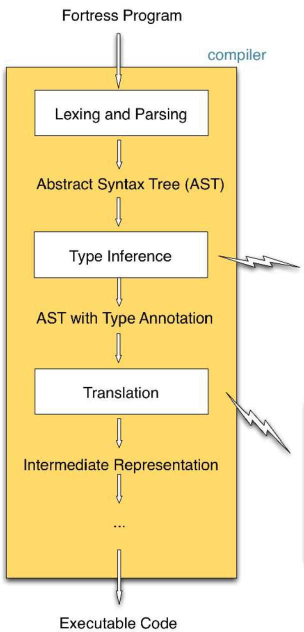
Result: The spacecraft was lost. The project cost was $327.6 million for both orbiter and lander.

Patriot Missile Failure

Accumulated rounding error in patriot missile software caused a missile to track its target incorrectly.

Result: SCUD missile was able to strike an army barrack, resulting in 28 Americans killed.

Formalized Semantics



- Provides unambiguous specification for compiler writers
  - Fewer insidious bugs
  - More portable code

Typing Rules

Expression typing: $p;\Delta;\Gamma \vdash e : \tau$

[T-Var] $\quad p;\Delta;\Gamma \vdash x : \Gamma(x)$

[T-Self] $\quad p;\Delta;\Gamma \vdash \texttt{self} : \Gamma(\texttt{self})$

[T-Object] $$\frac{\texttt{object}\ O\ \_\ (\overrightarrow{\_:\tau',})\ \_ \in p \qquad p;\Delta \vdash O[\overrightarrow{\tau,}]\ \text{ok} \quad p;\Delta;\Gamma \vdash \overrightarrow{e} : \overrightarrow{\tau''} \quad p;\Delta \vdash \overrightarrow{\tau''} <: \overrightarrow{\tau'}}{p;\Delta;\Gamma \vdash O[\overrightarrow{\tau,}]\,(\overrightarrow{e,}) : O[\overrightarrow{\tau,}]}$$

[T-Field] $$\frac{p;\Delta;\Gamma \vdash e_0 : \tau_0 \quad bound_\Delta(\tau_0) = O[\overrightarrow{\tau,}] \qquad \texttt{object}\ O[\overrightarrow{\alpha \triangleleft \_,}]\ \_\ \overrightarrow{x:\tau' = e;}\ \_ \in p}{p;\Delta;\Gamma \vdash e_0.x_i : [\overrightarrow{\tau/\alpha}]\tau'_i}$$

[T-Method] $$\frac{p;\Delta;\Gamma \vdash e_0 : \tau_0 \quad mtype_p(f, bound_\Delta(\tau_0)) = \{[\overrightarrow{\alpha \triangleleft N,}]\ \overrightarrow{\tau'} \to \tau'_0\} \qquad p;\Delta \vdash \overrightarrow{\tau}\ \text{ok} \quad p;\Delta \vdash \overrightarrow{\tau} <: [\overrightarrow{\tau/\alpha}]\overrightarrow{N} \qquad p;\Delta;\Gamma \vdash \overrightarrow{e} : \overrightarrow{\tau''} \quad p;\Delta \vdash \overrightarrow{\tau''} <: [\overrightarrow{\tau/\alpha}]\overrightarrow{\tau'}}{p;\Delta;\Gamma \vdash e_0.f[\overrightarrow{\tau,}]\,(\overrightarrow{e,}) : [\overrightarrow{\tau/\alpha}]\tau'_0}$$

Evaluation Rules

Evaluation rules: $p \vdash E[R] \longrightarrow E[e]$

[R-Field] $$\frac{\texttt{object}\ O[\overrightarrow{\alpha \triangleleft \_,}]\,(\overrightarrow{x':\_,})\ \_\ \overrightarrow{x:\_ = e;}\ \_ \in p}{p \vdash E[O[\overrightarrow{\tau,}]\,(\overrightarrow{v,}).x_i] \longrightarrow E[[\overrightarrow{\tau/\alpha}][\overrightarrow{v/x'}]e_i]}$$

[R-Method] $$\frac{\texttt{object}\ O\ \_\ (\overrightarrow{x':\_,})\ \_ \in p \qquad mbody_p(f[\overrightarrow{\tau',}], O[\overrightarrow{\tau,}]) = \{(\overrightarrow{x}) \to e\}}{p \vdash E[O[\overrightarrow{\tau,}]\,(\overrightarrow{v,}).f[\overrightarrow{\tau',}]\,(\overrightarrow{v',})] \longrightarrow E[[\overrightarrow{v/x'}][O[\overrightarrow{\tau,}]\,(\overrightarrow{v,})/\texttt{self}][\overrightarrow{v'/x}]e]}$$

- Allows proofs of soundness and formal analysis

Type Soundness Proof

Theorem (Subject Reduction). If p is well-typed, $p;\Delta;\Gamma \vdash e : \tau$, and $p \vdash e \longrightarrow e'$ then $p;\Delta;\Gamma \vdash e' : \tau'$ where $p;\Delta \vdash \tau' <: \tau$.

Proof. The proof is by case analysis on the evaluation rule applied.

Case [R-Field]: $e = E[O[\overrightarrow{\tau,}]\,(\overrightarrow{v',}).x_i]$
$e' = E[[\overrightarrow{\tau/\alpha}][\overrightarrow{v'/x'}]e_i]$

By the well-typedness of e, we have $p;\Delta;\Gamma \vdash O[\overrightarrow{\tau,}]\,(\overrightarrow{v',}).x_i : [\overrightarrow{\tau/\alpha}]\tau''_i$ where $\texttt{object}\ O[\overrightarrow{\alpha \triangleleft N,}]\,(\overrightarrow{x':\tau',}) \triangleleft \{\overrightarrow{M,}\}\ \overrightarrow{x:\tau'' = e;}\ \overrightarrow{fd;}\ \texttt{end} \in p$.

By typing rules [T-Object], [T-ObjectDef], [T-FieldDef], and [W-Both], we have:

(1a) $p;\Delta;\Gamma \vdash \overrightarrow{v'} : \overrightarrow{\tau_v}$
(1b) $p;\Delta \vdash \overrightarrow{\tau_v} <: \overrightarrow{\tau'}$
(2a) $p;\overrightarrow{\alpha <: N};\overrightarrow{x':\tau'} \vdash e_i : \tau'''_i$
(2b) $p;\overrightarrow{\alpha <: N} \vdash \tau'''_i <: \tau''_i$
(3b) $p;\Delta \vdash \overrightarrow{\tau} <: [\overrightarrow{\tau/\alpha}]\overrightarrow{N}$
(4a) $p;\Delta;\Gamma \vdash O[\overrightarrow{\tau,}]\,(\overrightarrow{v',}) : O[\overrightarrow{\tau,}]$

By the Weakening Lemma and the Term Substitution Lemma applied to (2a), (1a), and (1b), we have:

(5a) $p;\Delta,\overrightarrow{\alpha <: N};\Gamma \vdash [\overrightarrow{v'/x'}]e_i : \tau''''_i$
(5b) $p;\Delta,\overrightarrow{\alpha <: N} \vdash \tau''''_i <: \tau'''_i$

By the Type Substitution Lemma applied to (5a) and (3b), we have:

(6a) $p;\Delta;[\overrightarrow{\tau/\alpha}]\Gamma \vdash [\overrightarrow{\tau/\alpha}][\overrightarrow{v'/x'}]e_i : \tau'''''_i$
(6b) $p;\Delta \vdash \tau'''''_i <: [\overrightarrow{\tau/\alpha}]\tau''''_i$

By the Weakening Lemma, the Type Substitution Lemma, and [S-Trans], we have:

(7b) $p;\Delta \vdash \tau'''''_i <: [\overrightarrow{\tau/\alpha}]\tau''_i$

By applying the Replacement Lemma to judgements (7a) and (8b), we finish the case.

Case [R-Method]: $\cdots$ $\square$

Example Program in Fortress

```
object Main[]() traits {Object}
  myself:Main[] = self
  identity[](x:Object):Object = x
end

Main[]().identity[](Main[]().myself)
```

Mechanized Semantics

- Tests soundness of language semantics

Soundness of the Example Program

Suppose p is the example program.

If $p;\emptyset;\emptyset \vdash$ `Main[]().identity[](Main[]().myself)` : `Object`
and $p \vdash$ `Main[]().identity[](Main[]().myself)` $\longrightarrow^*$ `Main[]()`
then $p;\emptyset;\emptyset \vdash$ `Main[]()` : `Main[]` where $p;\emptyset \vdash$ `Main[]` $<:$ `Object`.

Reductions of the Example Program using PLT Redex

```
((invk
  (Main (tas) (vas))
  (identity
    (tas)
    (vas
     (invk
      (Main (tas) (vas))
      (myself)))))
 Object)
```

→

```
((invk
  (Main (tas) (vas))
  (identity
    (tas)
    (vas (Main (tas) (vas)))))
 Object)
```

→

```
((Main (tas) (vas))
 (Main (tas)))
```

Formalizing Language Semantics

- Provides unambiguous specification for compiler writers.
  - > Fewer insidious bugs
  - > More portable code
- Allows proofs of soundness and formal analysis.
  - > "Well-typed programs do not go wrong."
  - > Catch errors at compile time to avoid run-time disasters (Ariane 5, Mars Climate Orbiter, Patriot Missile Failure).

Fortress Type System

- Our static type system can encode data types usually considered the province of dynamic type systems.
- We have completed soundness proofs for the associated type calculi.
- Algebraic properties drive implementation strategies to achieve mix-and-match code selection.

Types Example: Data Types

$$
\begin{array}{l}
\texttt{value trait } \mathrm{List}[\![T \texttt{ extends } U]\!] \texttt{ extends } \mathrm{List}[\![U]\!] \texttt{ where } \{U \texttt{ extends } \mathrm{Object}\} \\
\quad \texttt{excludes } \{T\} \\
\quad \texttt{comprises } \{\mathrm{Empty}, \mathrm{Cons}[\![T]\!]\} \\
\quad \mathit{cons}(\mathit{first}' \colon U, \texttt{self}) \colon \mathrm{List}[\![U]\!] = \mathrm{Cons}(\mathit{first}', \texttt{self}) \\
\quad \mathit{append}(\texttt{self}, \mathit{rest}' \colon \mathrm{List}[\![U]\!]) \colon \mathrm{List}[\![U]\!] \\
\texttt{end}
\end{array}
$$

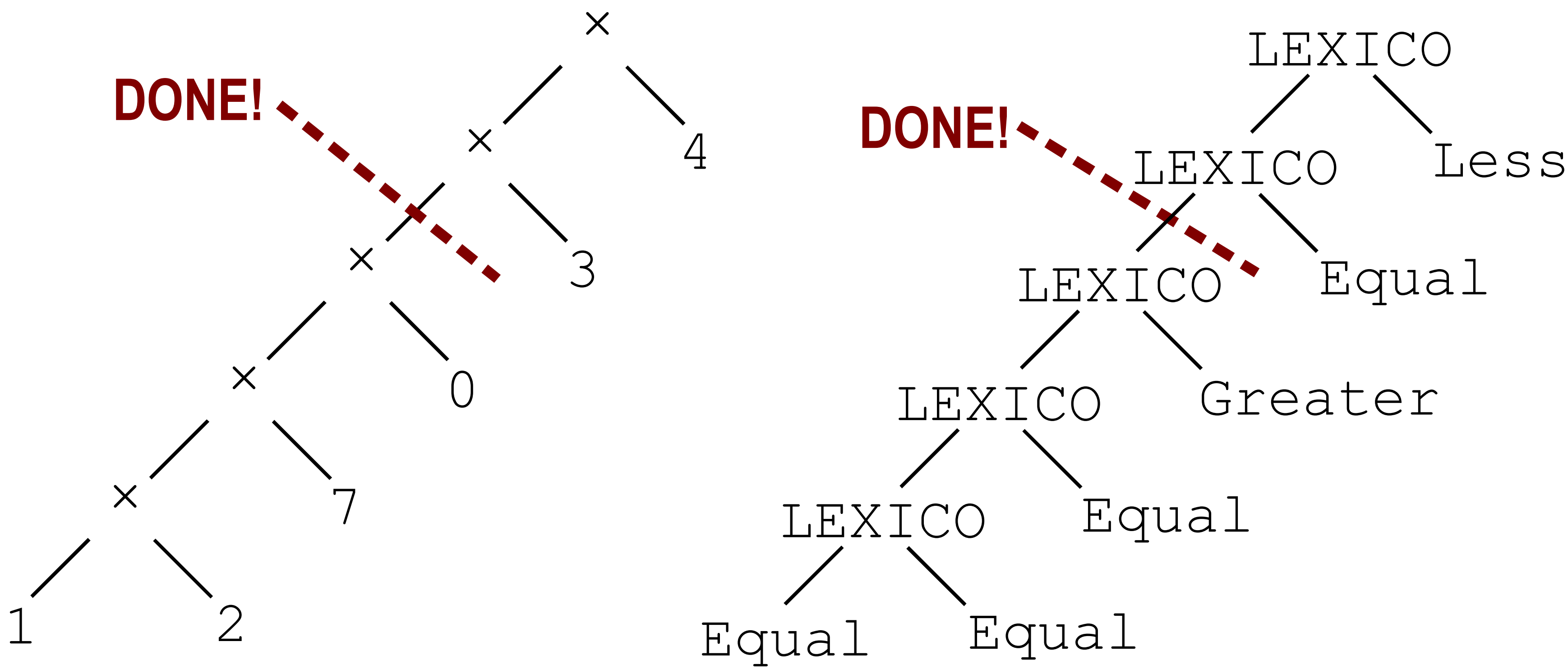
$$
\begin{array}{l}
\texttt{value object } \mathrm{Empty} \texttt{ extends } \mathrm{List}[\![T]\!] \texttt{ where } \{T \texttt{ extends } \mathrm{Object}\} \\
\quad \mathit{append}(\texttt{self}, \mathit{rest}' \colon \mathrm{List}[\![T]\!]) \colon \mathrm{List}[\![T]\!] = \mathit{rest}' \\
\texttt{end}
\end{array}
$$

$$
\begin{array}{l}
\texttt{value object } \mathrm{Cons}[\![T \texttt{ extends } U]\!](\mathit{first} \colon T, \mathit{rest} \colon \mathrm{List}[\![T]\!]) \texttt{ extends } \mathrm{List}[\![U]\!] \\
\quad \texttt{where } \{U \texttt{ extends } \mathrm{Object}\} \\
\quad \mathit{append}(\texttt{self}, \mathit{rest}' \colon \mathrm{List}[\![U]\!]) \colon \mathrm{List}[\![U]\!] = \mathit{cons}(\mathit{first}, \mathit{append}(\mathit{rest}, \mathit{rest}')) \\
\texttt{end}
\end{array}
$$

Types Example: Algebraic Properties

```
value trait Comparison
    extends { IdentityEquality⟦Comparison⟧,
              Associative⟦Comparison, LEXICO⟧,
              HasIdentity⟦Comparison, LEXICO⟧,
              HasLeftZeroes⟦Comparison, LEXICO, isLeftZeroForLEXICO⟧ }
    comprises { TotalComparison, Unordered }
  opr LEXICO(self, other: Comparison): Comparison
  isLeftZeroForLEXICO(self): Boolean
  opr ≡(self, other: Comparison): Boolean
  getter hashCode(): ℤ64
  toString(): String
end
```

Zeroes Can Stop Iteration Early



Outline

- Fortress Programming Language
- Formalism in Fortress
- Project Fortress

Design Strategy

- Devise a specification, implementation, formal semantics, and library code in parallel.
- Each provides different insights into the language.
- Each provides feedback to the others.

Status

- Draft specification and preliminary open source release available
- BSD license
- http://research.sun.com/projects/plrg

Fostering Community Development

- An effective language needs good compilers, tools, development environments, libraries, tutorials.
- An effective language should belong to the community.
- An effective language should be *built* by the community.

Establishing an Open Source Community

- Establish open source projects as enabling technologies.
- Provide initial code and participate in extensions.
- Establish Cooperative Research agreements with external teams (in academia, industry, non-profits).

sukyoung.ryu@sun.com

http://research.sun.com/projects/plrg