Project Fortress:
A Growable Language for Scientists and Engineers

Sukyoung Ryu

Department of Computer Science
Korea Advanced Institute of Science and Technology

October 10, 2010

Project Fortress

- A multicore language for scientists and engineers

Project Fortress

- A multicore language for scientists and engineers
- Run your whiteboard in parallel!

Project Fortress

- A multicore language for scientists and engineers
- Run your whiteboard in parallel!

$$v_{\mathrm{norm}} = v / \|v\|$$

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

$$C = A \cup B$$

$$y = 3x \sin x \cos 2x \log \log x$$

Project Fortress

- A multicore language for scientists and engineers
- Run your whiteboard in parallel!

$$v_{\text{norm}} = \underline{v/\underline{\|v\|}}$$

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

$$C = \underline{A \cup B}$$

$$y = \underline{\underline{3x}\ \underline{\sin x}}\ \underline{\cos \underline{2x}}\ \underline{\log \underline{\log x}}$$

Project Fortress

- A multicore language for scientists and engineers
- Run your whiteboard in parallel!

$$v_{\text{norm}} = \underline{v / \underline{\|v\|}}$$

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

$$C = \underline{A \cup B}$$

$$y = \underline{\underline{3x}\ \underline{\sin x}}\ \underline{\cos \underline{2x}}\ \underline{\log \underline{\log x}}$$

- "Growing a Language"

  Guy L. Steele Jr., keynote talk, OOPSLA 1998

  Higher-Order and Symbolic Computation 12, 221-236 (1999)

Project Fortress

- Fortress is a growable, mathematically oriented, parallel programming language for scientific applications.
- Started under Sun/DARPA HPCS program, 2003–2006.
- Fortress is an open-source project with international participation.
- The Fortress 1.0 release (March 2008) synchronized the specification and implementation.
- Now at Sun Labs, Oracle, we are growing the language and libraries and developing a compiler.

Mathematical Syntax

Mathematical Syntax

Integrated mathematical and object-oriented notation

- Supports a stylistic spectrum that runs from Fortran to Java™—and sticks out at both ends!
  - > More conventionally mathematical than Fortran
    - ∗ Compare `a*x**2+b*x+c` and $a\,x^2 + b\,x + c$
  - > More object-oriented than Java
    - ∗ Multiple inheritance
    - ∗ Numbers, booleans, and characters are objects
  - > To find the size of a set S: either $|S|$ or $S.size$
    - ∗ If you prefer $\#S$, defining it is a one-liner.

Mathematical Syntax Using Unicode

- Full Unicode character set available for use, including mathematical operators and Greek letters:

$\times$	$\div$	$\oplus$	$\ominus$	$\otimes$	$\oslash$	$\odot$	$\approx$	α	β	γ	δ
$\boxplus$	$\boxminus$	$\boxtimes$	$\leftrightarrow$	$\wedge$	$\vee$	$\equiv$	$\not\equiv$	ϵ	ζ	η	θ
$\leq$	$\geq$	$\sum$	$\prod$	$\prec$	$\preccurlyeq$	$\succcurlyeq$	$\succ$	ι	κ	λ	μ
$\cap$	$\cup$	$\uplus$	$\subset$	$\subseteq$	$\supseteq$	$\supset$	$\in$	ξ	π	ρ	σ
$\sqcap$	$\sqcup$	$\sqsubset$	$\sqsubseteq$	$\sqsupseteq$	$\sqsupset$	$\neg$	$\notin$	ϕ	χ	ψ	ω
$\lfloor$	$\rfloor$	$\lceil$	$\rceil$	$\langle$	$\rangle$	$\curlywedge$	$\curlyvee$	Γ	Θ	and so on	

- Use of "funny characters" is under the control of libraries (and therefore users)

Mathematical Syntax Example

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: $\|r\| = \|x - A\,z\|$

KAIST Computer Science

Mathematical Syntax Example

NAS NPB 2.3 Serial Code in Fortran

```
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 )
```

Mathematical Syntax Example

NAS NPB 1 Specification

$$
\begin{aligned}
&z = 0\\
&r = x\\
&\rho = r^T r\\
&p = r\\
&\textbf{DO } i = 1, 25\\
&\qquad q = A\, p\\
&\qquad \alpha = \rho / (p^T q)\\
&\qquad z = z + \alpha\, p\\
&\qquad \rho_0 = \rho\\
&\qquad r = r - \alpha\, q\\
&\qquad \rho = r^T r\\
&\qquad \beta = \rho / \rho_0\\
&\qquad p = r + \beta\, p\\
&\textbf{ENDDO}\\
&\text{compute residual norm explicitly: } \|r\| = \|x - A\, z\|
\end{aligned}
$$

in Fortress

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

Mathematical Syntax by 'Fortify'

- Emacs-based code formatter
- Fortress programs can be typed on ASCII keyboards.
- Code automatically formatted for processing by LaTeX

$$sum\colon \mathbb{R}64 := 0$$

$$\texttt{for}\ k \leftarrow 1 : n\ \texttt{do}$$

$$\quad a_k := (1 - \alpha) b_k$$

$$\quad sum \mathrel{+}= c_k\, x^k$$

$$\texttt{end}$$

All code on these slides was formatted by this tool.

Mathematical Syntax by 'Fortify'

- Emacs-based code formatter
- Fortress programs can be typed on ASCII keyboards.
- Code automatically formatted for processing by LaTeX

$sum\colon \mathbb{R}64 := 0$

$\texttt{for}\ k \leftarrow 1 : n\ \texttt{do}$

$\quad a_k := (1 - \alpha) b_k$

$\quad sum \mathrel{+}= c_k\, x^k$

$\texttt{end}$

```
sum: RR64 := 0
for k<-1:n do
   a[k] := (1-alpha)b[k]
   sum += c[k] x^k
end
```

All code on these slides was formatted by this tool.

Mathematical Syntax Using Editors

- Fortress mode for Emacs
  - Provides syntax coloring
  - Some automatic formatting
  - Unicode font conversion
- Fortress NetBeans™ plug-in
  - Syntax highlighting
  - Mark occurrences
  - Instant rename
- Fortress Eclipse plug-in (work in progress)
- These tools were contributed by people outside Sun.

Parallelism by Default

A Parallel Language

High productivity for multicore, SMP, and cluster computing

- Hard to write a program that isn't potentially parallel
- Support for parallelism at several levels
  - Expressions
  - Loops, reductions, and comprehensions
  - Parallel code regions
  - Explicit multithreading
- Shared global address space model with shared data
- Thread synchronization through atomic blocks and transactional memory

Implicit Parallelism

- Tuples

$$(a, b, c) = \big(f(x), g(y), h(z)\big)$$

- Functions, operators, method call recipients, and their arguments

$e_1\, e_2$ $\qquad$ $e_1 + e_2$

$e_1(e_2)$ $\qquad$ $e_1.method(e_2)$

- Expressions with generators

$$s = \sum_{k \leftarrow 1:n} c_k\, x^k$$

$$\{\, x^2 \mid x \leftarrow xs, x > 43 \,\}$$

for $k \leftarrow 1:n$ do

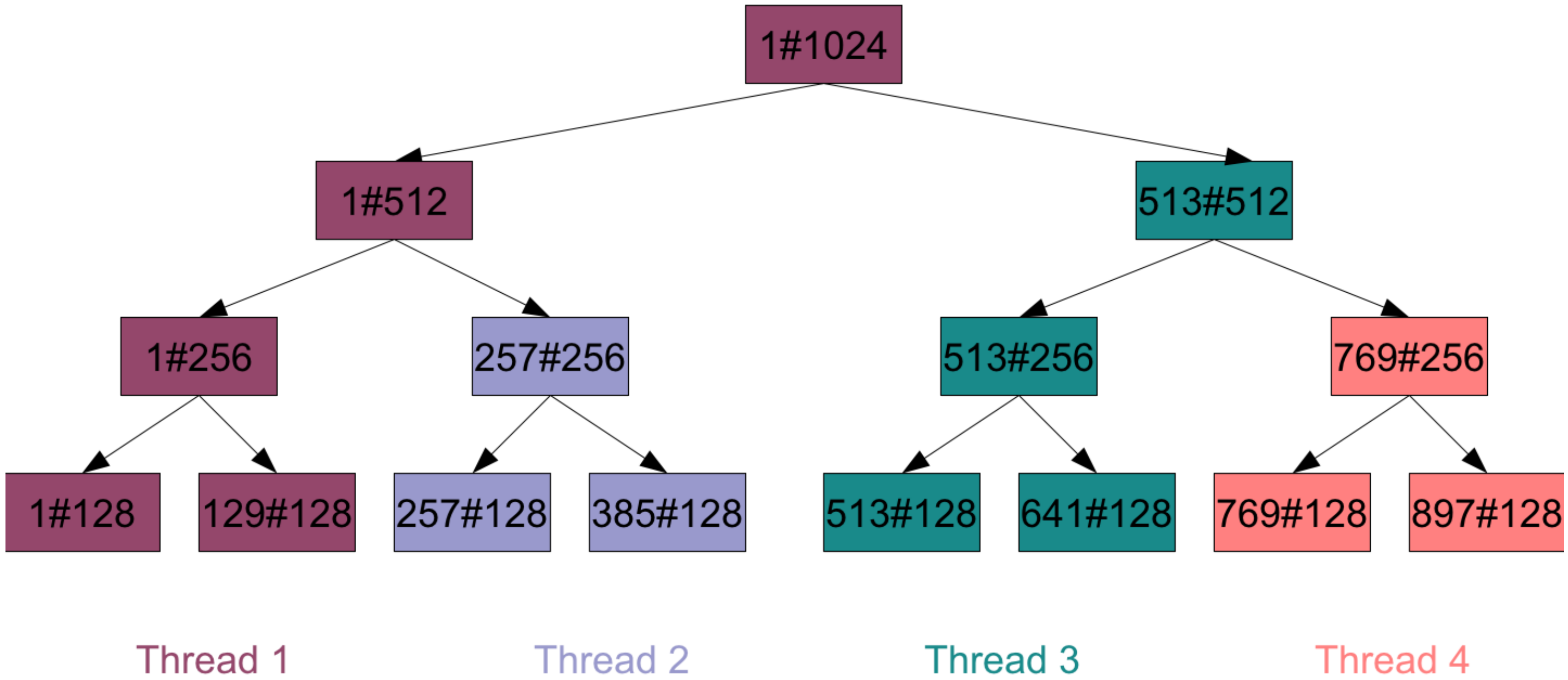
$\quad sum \mathrel{+}= c_k\, x^k$

end

$$sum \mathrel{+}= c_k\, x^k, k \leftarrow 1:n$$

Recursive Subdivision and Work Stealing

$$\sum_{k \leftarrow 1\#1024} c_k \, x^k$$



Should Parallelism Be the Default?

- "Loop" is a misleading term in the Fortress world
  - A set of executions of a parameterized block of code.
  - Whether and how they run in parallel is a separate concern.
- In Fortress loops are parallel by default
  - This is a convention of library code, as we will see.

In order to get programmers to write parallel code, we must change their (and our) mind set. The ubiquitous parallelism of Fortress is our attempt.

Growable Language

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.

Growing a Language

"So I think the sole way to win
is to plan for growth
with help from users."

Guy L. Steele Jr.
keynote talk, OOPSLA 1998;
Higher-Order and Symbolic Computation 12, 221-236 (1999)

Design Strategy

Consider how a proposed language feature might be provided by a library rather than building features directly into the compiler.

This requires control over both syntax and semantics, not just the ability to add new functions and methods.

`for` Loops in Fortress

```
for generators do
    body
end
```

- Generator list produces data items (usually in parallel).
- Body computation is executed for each data item.
- Whether/how they run in parallel is defined in the libraries.

`for` Loops in Fortress

```
for generators do
    body
end
```

```
for i ← {1,2}, j ← {3,4} do
    println "(" i ", " j ")"
end
```

- Generator list produces data items (usually in parallel).
- Body computation is executed for each data item.
- Whether/how they run in parallel is defined in the libraries.
- In Fortress, `for` loops are parallel by default.

$(2, 3)$

$(1, 3)$

$(2, 4)$

$(1, 4)$

Desugaring `for` Loops

$$g_1 = \{1, 2\}$$
$$g_2 = \{3, 4\}$$

```
for i ← g1, j ← g2 do
    println "(" i ", " j ")"
end
```

Desugaring `for` Loops

$$g_1 = \{1, 2\}$$
$$g_2 = \{3, 4\}$$

$\texttt{for}\ i \leftarrow g_1, j \leftarrow g_2\ \texttt{do}$

$\quad println\ \text{“(”}\ i\ \text{“, ”}\ j\ \text{“)”}$

$\texttt{end}$

desugars to

$g_1.loop(\texttt{fn}\ i \Rightarrow$

$\quad g_2.loop(\texttt{fn}\ j \Rightarrow$

$\quad\quad println\ \text{“(”}\ i\ \text{“, ”}\ j\ \text{“)”}\ ))$

Desugaring `for` Loops

- Design strategy

  Consider how a proposed language feature might be provided by a library rather than building features directly into the compiler.

Desugaring `for` Loops

- Design strategy

  Consider how a proposed language feature might be provided by a library rather than building features directly into the compiler.

- `for` loops desugaring in the current implementation is built directly into the interpreter.

Desugaring `for` Loops

- Design strategy

  Consider how a proposed language feature might be provided by a library rather than building features directly into the compiler.

- `for` loops desugaring in the current implementation is built directly into the interpreter.
- `for` loops desugaring by syntactic abstraction is provided by a library.

Syntactic Abstraction Goals[a]

- New syntax indistinguishable from the core syntax
- Similar syntax for definition/use of a language extension
- Composition of independent language extensions
- Expansion into other language extensions
- Mutually recursive definition of a language extension

[a] Growing a Syntax, Eric Allen, Ryan Culpepper, Janus Dam Nielsen, Jon Rafkind, and Sukyoung Ryu. The Foundations of Object-Oriented Languages 2009

KAIST Computer Science

Grammar of Simplified `for` Loops

```
grammar ForLoop extends { Expression, Identifier }
 Expr |:=
    for { i : Id ← e : Expr , ? Space }* do  block : Expr end ⇒
    ⊲[ for₂ i ∗∗; e ∗∗; do  block; end ]⊳
 | for₂ i : Id∗; e : Expr∗; do  block : Expr; end ⇒
   case  i of
     Empty ⇒ ⊲[ block ]⊳
     Cons(ia, ib) ⇒
       case  e of
         Cons(ea, eb) ⇒
           ⊲[ ((ea).loop(fn  ia ⇒ (for₂ ib ∗∗; eb ∗∗;
                                          do  block; end)))]⊳
     end
   end
end
```

Syntactic Abstraction in Fortress

- New syntax indistinguishable from the core syntax

$$\texttt{for}\ i \leftarrow g_1, j \leftarrow g_2\ \texttt{do}$$
$$\quad println\ \text{“ (” } i \text{ “, ” } j \text{ “) ”}$$
$$\texttt{end}$$

- Similar syntax for definition/use of a language extension

$$\texttt{for}\ \{\, i{:}\mathrm{Id} \leftarrow e{:}\mathrm{Expr}\ ,?\,\mathrm{Space}\,\} * \texttt{do}\ block{:}\mathrm{Expr}\ \texttt{end}\ \Rightarrow \ldots$$

KAIST Computer Science

Syntactic Abstraction in Fortress

- Composition of independent macros

`grammar` ForLoop `extends` { Expression, Identifier }

- Expansion into other language extensions

$$\text{Expr} \mid := $$
$$\texttt{for}\ \{\, i : \text{Id} \leftarrow e : \text{Expr}\ ,?\ \text{Space}\,\}{*}\ \texttt{do}\ \ block : \text{Expr}\ \texttt{end} \Rightarrow$$
$$\langle\!\lbrack\, for_2\ i**; e**; \texttt{do}\ \ block; \texttt{end}\,\rbrack\!\rangle$$
$$\mid\ for_2\ i : \text{Id}{*}; e : \text{Expr}{*}; \texttt{do}\ \ block : \text{Expr}; \texttt{end} \Rightarrow$$
$$\ldots$$

Syntactic Abstraction in Fortress

- Mutually recursive definition of a language extension

$$
\begin{array}{l}
\mid \mathit{for}_2\ i : \mathrm{Id}*; e : \mathrm{Expr}*; \texttt{do}\ \ \mathit{block} : \mathrm{Expr}; \texttt{end} \Rightarrow \\
\quad \texttt{case}\ \ i\ \texttt{of} \\
\qquad \mathrm{Empty} \Rightarrow \lhd\!\!\left[\ \mathit{block}\ \right]\!\!\rhd \\
\qquad \mathrm{Cons}(ia, ib) \Rightarrow \\
\qquad\quad \texttt{case}\ \ e\ \texttt{of} \\
\qquad\qquad \mathrm{Cons}(ea, eb) \Rightarrow \\
\qquad\qquad\quad \lhd\!\!\left[\ ((ea).\mathit{loop}(\texttt{fn}\ \ ia \Rightarrow (\mathit{for}_2\ ib**; eb**; \right. \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \left. \texttt{do}\ \ \mathit{block}; \texttt{end})))\right]\!\!\rhd \\
\qquad \texttt{end} \\
\quad \texttt{end}
\end{array}
$$

Desugaring `for` Loops Recursively

`for` $i \leftarrow g_1, j \leftarrow g_2$ `do` *println* "(" i ", " j ")" `end`

Desugaring `for` Loops Recursively

`for` $i \leftarrow g_1, j \leftarrow g_2$ `do` $println$ "(" i ", " j ")" `end`

desugars to

$g_1.loop($`fn` $i \Rightarrow$
    `for` $j \leftarrow g_2$ `do`
        $println$ "(" i ", " j ")"
    `end`$)$

Desugaring `for` Loops Recursively

`for` $i \leftarrow g_1, j \leftarrow g_2$ `do` $\textit{println}$ "(" i ", " j ")" `end`

desugars to

$g_1.loop($`fn` $i \Rightarrow$
`for` $j \leftarrow g_2$ `do`
$\textit{println}$ "(" i ", " j ")"
`end`$)$

desugars to

$g_1.loop($`fn` $i \Rightarrow$
$g_2.loop($`fn` $j \Rightarrow$
$\textit{println}$ "(" i ", " j ")" $))$

Syntax Normalization

- Parsing stage

  transforms a source program (in string) into *a parsed program* (in *node expression*)

- Transformation stage

  transforms the parsed program into a program in core Fortress AST

Parsing: Source Program ⇒ Parsed Program

- First step

  parses the macro definition into an intermediate form to generate a parser that recognizes the new syntax.

- Second step

  uses the generated parser to parse a source program using the new syntax.

Node Expressions

$$
\begin{array}{lcl}
\mathit{NodeExpr} & ::= & \mathit{PatternVar} \\
& | & \mathit{Transformer}\ (\overline{\mathit{NodeExpr}}) \\
& | & \mathit{NodeConstructor}(\overline{\mathit{NodeExpr}}) \\
& | & \mathrm{Ellipses}(\mathit{NodeExpr}, \overline{\mathit{NodeExpr}}) \\
& | & \texttt{case}\ \mathit{PatternVar}\ \texttt{of} \\
& & \quad \mathrm{Empty} \Rightarrow \mathit{NodeExpr} \\
& & \quad \mathrm{Cons}(\mathit{PatternVar}, \mathit{PatternVar}) \Rightarrow \mathit{NodeExpr} \\
& & \texttt{end}
\end{array}
$$

Transformation: Parsed Program ⇒ AST

- Pattern variables are substituted by the corresponding inputs.
- Transformers are replaced with their bodies, substituting pattern variables along the way.
- Core Fortress AST nodes transform their arguments.
- Ellipses nodes transform multiple occurrences of patterns.
- Case expressions match input to a constructor and invoke the corresponding transformer.

KAIST Computer Science

Translation of Node Expressions

[Pattern Variable]

$$\frac{\Gamma(x) = v}{\Upsilon, \Gamma \vdash x \rightarrow \Upsilon, \Gamma \vdash v}$$

[Node Constructor]

$$\frac{\Upsilon, \Gamma \vdash \overline{n} \rightarrow \Upsilon, \Gamma \vdash \overline{n'}}{\Upsilon, \Gamma \vdash c(\overline{n}) \rightarrow \Upsilon, \Gamma \vdash c(\overline{n'})}$$

[Macro Invocation Arguments]

$$\frac{\Upsilon, \Gamma \vdash \overline{n} \rightarrow \Upsilon, \Gamma \vdash \overline{n'}}{\Upsilon, \Gamma \vdash t(\overline{n}) \rightarrow \Upsilon, \Gamma \vdash t(\overline{n'})}$$

[Macro Invocation]

$$\frac{\Upsilon(t) = t\ \overline{x}.n}{\Upsilon, \Gamma \vdash t(\overline{v}) \rightarrow \Upsilon, \Gamma\left[\overline{x} \mapsto \overline{v}\right] \vdash n}$$

Typed Macros (Work in Progress)

- Soudness

  If a macro definition is well typed then there are no type errors at the use sites of the macro unless the user of the macro provides an input of a wrong type.

Designed to Grow

Technical design supports growth by an open-source community.

- Emphasis on replaceable components with multiple versions
- Language extensibility
  - > Parametric polymorphism with multiple inheritance
  - > Overloading of functions, methods, and operators
  - > User-defined syntactic extensions
- Plenty of room for experimentation
- Language encourages unit testing and explicit descriptions of code invariants and properties.

Summary

- Fortress is a growable language.
- Syntactic abstraction supports the language growth.
- Implementation is available from the website:
  http://projectfortress.sun.com/

Sukyoung Ryu

sryu@cs.kaist.ac.kr
http://plrg.kaist.ac.kr