Fortress

Kannan Goundan
Stephen Kou
Fernando Pereira
This Presentation

- The history of Fortress
- General language features
- Parallel processing features
- Demonstration
Part 1

THE HISTORY OF FORTRESS
Background and Status

• Developed by Sun Microsystems for the DARPA high-performance computing initiative.
  – Didn’t make it to Phase III
• Spec is at version “1.0 beta”
• Still being developed as an open source project. Mailing list is active.
• Implementation is weak.
  – Still only an unoptimized interpreter.
  – No static checking (undefined variables, type checking)
  – Many of the parallel features aren’t implemented.
Philosophy

• “Do for Fortran what Java did for C”
• Guy Steele is one of the designers
  – Co-creator of Scheme, worked on Java spec
  – “Growing a Language” (talk at OOPSLA ’98)
• Initially targeting scientific computing, but meant to be usable for anything.
• Designed from scratch.
Part two

GENERAL LANGUAGE FEATURES
Readability

• You can use tons of Unicode symbols.
  – Each has an ASCII equivalent.
• Mathematical syntax. What you write on the blackboard works.
• Minimize clutter
  – Don’t specify types that can be inferred.
  – Get rid of noisy punctuation (semicolons).
• Two input modes (Unicode vs ASCII). An additional typeset output mode.
Operators

- The “popular” operators:
  
  \[\begin{align*}
  + & \quad - \quad / \quad \bowtie = \quad < \quad > \quad | \quad \{ \quad \}
  \end{align*}\]

- Abbreviated operators:
  
  \[\begin{align*}
  [\big\{} \quad \big\} \quad \bowtie \quad =/= \quad >= \quad -> \quad \Rightarrow \quad |-> \quad <| \quad |> \quad \{\quad \}\quad \neq \quad \ge \quad \rightarrow \quad \Rightarrow \quad \leftrightarrow \quad \langle \quad \rangle
  \end{align*}\]

- Short names in all caps:
  
  \[\begin{align*}
  OPLUS \quad DOT \quad TIMES \quad SQCAP \quad AND \quad OR \quad IN
  \end{align*}\]

- Named:
Identifiers

• Regular:

\[
\begin{align*}
a & \quad \text{zip} & \quad \text{trickOrTreat} & \quad \text{foobar} \\
a & \quad \text{zip} & \quad \text{trickOrTreat} & \quad \text{foobar}
\end{align*}
\]

• Formatted:

\[
\begin{align*}
a^3 & \quad _a & \quad a_\text{vec} & \quad _a\text{hat} & \quad a_\text{max} & \quad \text{foo_bar} \\
a^3 & \quad a & \quad \vec{a} & \quad \hat{a} & \quad a_{\text{max}} & \quad \overline{\text{foo}}
\end{align*}
\]

• Greek Letters:

\[
\begin{align*}
\text{alpha} & \quad \text{beta} & \quad \text{GAMMA} & \quad \text{DELTA} \\
\alpha & \quad \beta & \quad \Gamma & \quad \Delta
\end{align*}
\]

• Unicode Names: \texttt{HEBREW_ALEF} \texttt{א}

• Blackboard Font:
Mathematical Syntax

“What if we tried really hard to make the mathematical parts of program look like mathematics?”  - Guy L. Steele

• Multiplication and exponentiation.
  – $x^2 + 3y^2 = 0$

• Operator chains: $0 \leq i < j < 100$

• Reduction syntax
  – $\text{factorial}(n) = \prod_{i=1}^{n} i$

$$x^2 + 3 \cdot y^2 = 0$$

$$\text{factorial}(n) = \prod_{i=1:n} i$$
Aggregate Expressions

- Set, array, maps, lists:
  - \{2, 3, 5, 7\}
  - \[“France” →“Paris”, “Italy”→“Rome”\]
  - \langle0, 1, 1, 2, 3, 5, 8, 13\rangle

- Set, array, maps, lists:
  - \{x^2 \mid x \leftarrow \text{primes}\}
  - \[x^2 \rightarrow x^3 \mid x \leftarrow \text{fibs, } x < 1000\]
  - \langle x(x+1)/2 \mid x \leftarrow 1\#100 \rangle

- Matrices:
  \[
  \begin{bmatrix}
  1 & 0 \\
  0 & A
  \end{bmatrix}
  \quad
  \begin{bmatrix}
  1 & 0 \\
  0 & A
  \end{bmatrix}
  \]
Dimension and Units

• Numeric types can be annotated with units

\[ \text{kineticEnergy}(m: \mathbb{R} \text{ kg}_\_ \text{, } v: \mathbb{R} \text{ m}_\/_\text{s}_\_): \mathbb{R} \text{ kg}_\_ \text{ m}^\_2/\text{s}_\_2 = (m \text{ v}^\_2) / 2 \]

• Common dimensions and units are provided in fortress standard library, e.g: \text{kg}, \text{m}, \text{s}

• Static safety checks

• Ex.:
Some Whitespace Sensitivity

• Whitespace must agree with precedence
  – Error: \(a+b\ /\ c+d\)

• Parentheses are sometimes required:
  \(A+B \lor C\)
  – “+” and “\(\lor\)” have no relative precedence.

• Fractions: \(\frac{1}{2} \times \frac{1}{2}\)

• Subscripting \((a[m\ n])\) vs vector multiplication: \((a[m\ n])\)
Example Code (Fortress)

**ASCII**

```plaintext
do
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||)
end
```

**Unicode**

```plaintext
do
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||)
end
```
Example Code (Typeset Fortress)

\[
\begin{align*}
z &= 0 \\
r &= x \\
\rho &= r^T r \\
p &= r \\
\textbf{do} & \ i = 1, 25 \\
& \quad \ q = A \ p \\
& \quad \ \alpha = \rho / (p^T q) \\
& \quad \ z = z + \alpha \ p \\
& \quad \ \rho_0 = \rho \\
& \quad \ r = r - \alpha q \\
& \quad \ \rho = r^T r \\
& \quad \ \beta = \rho / \rho_0 \\
& \quad \ p = r + \beta \ p \\
\textbf{end}
\end{align*}
\]

\[
\begin{align*}
z : \text{Vec} &= 0 \\
r : \text{Vec} &= x \\
p : \text{Vec} &= r \\
\rho : \text{Elt} &= r^T r \\
\textbf{for} & \ j \leftarrow \textbf{seq} (1 : cgit_{\max}) \ \textbf{do} \\
& \quad \ q = A \ p \\
& \quad \ \alpha = \frac{\rho}{p^T q} \\
& \quad \ z := z + \alpha \ \rho \\
& \quad \ r := r - \alpha q \\
& \quad \ \rho_0 = \rho \\
& \quad \ \rho := r^T r \\
& \quad \ \beta = \frac{\rho}{\rho_0} \\
& \quad \ p := r + \beta \ p \\
\textbf{end}
\end{align*}
\]
Object Oriented

- Classes (declared with `object`)
- Fields
- Virtual methods
- Multiple inheritance with “traits”. Like Java interfaces.
Traits

• Similar to Java interfaces, but...
• May contain method declarations...
• In addition to method definitions, but...
• Do not contain fields.
• Can be multiply inherited.
Examples

trait Loc
  getter position() : (R, R)
  displace(nx:R, ny:R) : ()
end

object Circle(x:R, y:R, r:R) extends {Loc,Geom}
  position() = (x, y)
  displace(nx:R, ny:R) = do x += nx; y += ny end
  area() = r * r * 3.1416
end

trait Geom
  area() : R
  density(unitWeight:R) = unitWeight * area()
end
Multiple Inheritance

• Multiple inheritance is tricky… Ex.:

• Traits have the flattening property:
  – the semantics of a method is the same if it is implemented in a trait or in the class that extends that trait.
  – ambiguous calls are explicitly resolved.
Functional Programming

- Everything is an expression
- Immutable by default
  - “:=” for mutable variables
- Closures
  - Standard library uses higher-order functions pervasively

\[
\text{add1}(n: \mathbb{Z}): \mathbb{Z} = n + 1
\]
\[
\text{applyN}(f: \mathbb{Z} \rightarrow \mathbb{Z}, n: \mathbb{N}, x: \mathbb{Z}): \mathbb{Z} = \text{do}
  v: \mathbb{Z} = x
  \text{remaining: } \mathbb{N} = n
  \text{while } \text{remaining} > 0 \text{ do}
    v := f(v)
    \text{remaining} -= 1
  \text{end}
  v
\text{end}
\]
\[
\text{composeN}(f: \mathbb{Z} \rightarrow \mathbb{Z}, n: \mathbb{N}): \mathbb{Z} \rightarrow \mathbb{Z} = \text{if } (n = 0) \text{ then}
  fn(x: \mathbb{Z}) \Rightarrow x
\text{else}
  \text{base} = \text{composeN}(f, n-1)
  fn(x: \mathbb{Z}) \Rightarrow f(\text{base}(x))
\text{end}
\]

applyN(add1, 4, 3) (composeN(add1, 4))(3)
Functional Programming

- Tagged unions
- Pattern matching
- List comprehensions

\[
x = \langle 2, 4, 6, 8, 10 \rangle
\]

\[
x = \langle x \mid x \leftarrow 1:10, \text{iseven}(x) \rangle
\]

\[
\text{iseven}(x: \mathbb{Z}): \text{Bool} = \\
\quad x \mod 2 = 0
\]

\[
\text{trait List comprises} \ \{ \text{Cons, Nil} \} \\
\text{end}
\]

\[
\text{object} \ \text{Cons}(h: \mathbb{Z}, t: \text{List}) \ \text{extends} \ \text{List} \\
\quad \text{head: } \mathbb{Z} = h \\
\quad \text{tail: } \text{List} = t \\
\text{end}
\]

\[
\text{object} \ \text{Nil} \ \text{extends} \ \text{List} \\
\text{end}
\]

\[
\text{sum}(l: \text{List}) = \text{typecase } l \ \text{of} \\
\quad \text{List} \Rightarrow l.\text{head} + \text{sum}(l.\text{tail}) \\
\quad \text{Nil} \Rightarrow 0 \\
\text{end}
\]
Operator Overloading

• Can be alphanumeric: \( a \ \text{MAX} \ b \)
• Juxtaposition is overloadable (multiplication, string concatenation).
• Dangerous, but...
  – Library writer can exercise restraint.
  – Fortress has more operators to go around. They don’t get *over*-overloaded.
Defining Operators

```
object Complex (r:R, i:R)
    opr +(self, other:Complex):Complex =
        Complex(r + other.r, i + other.i)
    opr MULT(self, other:Complex):Complex =
        Complex(r other.r - i other.i, i other.r + r other.i)
    toString():String =
        "Real part = " r ", Imaginary part = " i
end

run(args:String...):() = do
    c1:Complex = Complex(1.5, 2.3)
    c2:Complex = Complex(4.5, -2.7)
    println(c1)
    println(c2)
    println(c1 + c2)
    println(c1 MULT c2)
end
```
(Pre/in/post)-fix Operators

opr \textit{MINUS}(m:Z, n:Z) = m - n
opr \textit{NEG}(m:Z) = -m
opr (n:Z)\textit{FAC} = \text{if } n \leq 1 \text{ then } 1 \text{ else } n (n-1)\textit{FAC}\ \text{end}

run(args: String...):() = do
    println(7 \textit{MINUS} 3)
    println(NEG 3)
    println((7)FAC)
end

Output:

Parsing tests/fernando/oprN.fss: 979 milliseconds
Static checking: 92 milliseconds
Read FortressLibrary.tfs: 970 milliseconds
4
-3
5040
finish runProgram
Program execution: 2807 milliseconds
Static Parameters

- Type parameters.
- Can place restrictions with “where” clauses.
- Unlike Java, can use the type information at runtime.

```plaintext
object Box[T](var e: T)
    where {T extends Equality}
    put(e': T): () = e := e'
    get(): T = e
    opr =(self, Box[T] o) =
        self.e = o.e
end

cast[T](x: Object): T =
    typecase x in
        T ⇒ x
    else ⇒ throw CastException
end
```
Static Parameters

```
object Box[T](var e: T)
  where {T extends Equality}
put(e': T): () = e := e'
get(): T = e
opr =(self, Box[T] o) =
  self.e = o.e
end
```

- Unlike C++, type checking is modular. All type restrictions must be declared.
- Like C++, the compiler can generate multiple specialized versions of the function.
Static Parameters

- Can parameterize on values.
  - int, nat, bool
  - dimensions and units

- Define mathematical properties by parameterizing on functions.

```plaintext
run[bool debug]() = do
  ... 
  if (debug) then
    sanityCheck()
  end
  ...
end
```

```plaintext
reduce[T,nam op](List[T] l)
  where
  {T extends Assoc[T,op]}
object Number extends
  Assoc[Number,opr +]
end
```
Programming by Contract

factorial(n:Z) requires n ≥ 0
  if n = 0 then 1
  else n factorial (n - 1)
end

• Function contracts consists of three optional parts:
  – requires, ensures and invariants
Ensuring Invariants

```plaintext
mangle(input:List)
    ensures sorted(result)
    provided sorted(input)
    invariant size(input) =
    if input ≠ Empty then
        mangle(first(input))
        mangle(rest(input))
    end
```
Properties and Tests

• Invariants that must hold for all parameters:

```text
property isMonotonic =
∀(x:Z, y:Z)(x < y) → (f(x) < f(y))
```

• Tests consist of data plus code:

```text
test s:Set[Z] = {-1, 2, 3, 4}
test isMonS[x←s, y←s] =
    isMonotonic(x, y)
test isMon2[x←s, y←s] =
    isMonotonic(x, x^2 + y)
```
APIs and Components

• API
  – Interface of components;
  – only declarations, no definitions;
  – each API in the world has a distinct name;

• Components
  – Unit of compilation;
  – similar to a Java package;
  – components can be combined;
  – import and export APIs
APIs and Components

• Example:

```component Hello
  import print from IO
  export Executable
  run(args: String...) =
    print "Hello world" end
end

api IO
  print: String → ()
end

api Executable
  run(args: String...) → ()
end```
Part Three

PARALLELISM FEATURES
Reduction Variables

• For computing expressions as locally as possible, avoiding the need to synchronize when unnecessary.

• *Definition:* A variable $l$ is considered a reduction variable reduced using the reduction operator $\oplus$ for a particular thread group if it satisfies the following conditions:
  – Every assignment to $l$ within the thread group is of the form $l = e$, where exactly one operator or its group inverse is used
  – The value of $l$ is not otherwise read within the thread group.
  – The variable $l$ is not a free.
Threads

- Two types:
  - Implicit and Spawned (explicit) threads
- Five states:
  - Not started, executing, suspended, normal completion, abrupt completion
- Each thread has two components:
  - Body and execution environment
Implicit Threads

• Fortress has many constructs that lead to implicit thread creation:
  – Tuple expressions
  – also do blocks
  – Method invocations, function calls
  – for loops, comprehensions, sums, generated expressions, big operators
  – Extremum expressions
  – Tests
Implicit Threads

- Run as fork-join style: all threads created together, and all must complete before the expression completes.
- If any thread ends abruptly, the group as a whole will also end abruptly
  - Reduction variables should not be accessed after an abort.
- Programmer can not interact with implicit threads in any way. Generated by compiler.
- Fortress compiler may interleave the threads any way it likes.
  - The following code can run forever:

```fortress
r:z64:=0
(r:=1, while r=0 do end)
```
Explicit (spawned) Threads

• Created using the `spawn` expression.
• Programmer can interact with the thread explicitly; `spawn` returns an instance of `Thread[T]`, where `T` is the type of expression spawned
  – Can control with: `wait`, `ready`, `stop`
  – Accesses result with `val`.

```
T1 = spawn do e1 end
T2 = spawn do e2 end
A1 = T1.value()
A2 = T2.value()
```
Fortress’ Parallelism “Stack”

- **Highest Level**: Libraries to allocate locality-aware arrays
- **Library of Distributions**
- **at Expression**
- **Lowest Level**: Generators
Regions

- All threads, objects, array elements have an associated region.
- Obtained by calling `o.region` on object `o`.
- An abstract description of the machine
  - Forms the Region Hierarchy (a tree)
- Leaves of tree are mostly local (e.g. core in CPU).
- Near the root is more spread out (e.g. resources spread across entire cluster).
Arrays, Vectors, Matrices

- Assumed to be spread out across a machine
- Generally, Fortress will figure out where things go
  - For advanced users, they can manually combine, pivot, and redistribute arrays via the libraries.
- Each element may be in a different region
- Hierarchy of regions.
  - An element is local to its region, and all the enclosing regions in the hierarchy.
atomic Expression

atomic expr
tryatomic expr

• All IO will appear to happen simultaneously in a single step.
• Functions and methods can also be marked atomic.
• If an atomic expression ends abruptly, all writes are discarded.
• tryatomic throws an exception if it ends abruptly.
• Implicit threads may be spawned inside an atomic block, will complete before expression.
Abortable atomic

- Resembles a Transaction’s rollback
- Provides a user-level `abort()` that abandons the execution inside an atomic block

```plaintext
for i <- 1#100 do
    count += 1
end

for i <- 1#100 do
    atomic do
        count += 1
    end
end
```
Object Sharedness

- Regions described the location of an object on the machine
- Sharedness refers to the visibility of the object from other threads
- Basic rules of sharedness:
  - Reference objects are initially local
  - Sharedness can change with time
  - If an object is transitively reachable from more than one thread, it must be shared.
  - When a local object is stored into a shared object, it must be published (recursively).
  - Values of variables local to a thread must be published before they can be run in parallel with the parent thread.
Publishing local objects

• Publishing can be expensive
  – Publishing the root of a large nested object (e.g. a tree) will recursively publish all the children.
• Can cause short atomic expressions to take very long.
Distributions
at Expression

- A low-level construct giving the programmer the ability to explicitly place execution in a certain region

\[(v,w) = (a_i, \text{at a.region}(j) \text{ do } a_j \text{ end})\]

- Spawns two threads implicitly:
  - #1 calculated \(a_i\) locally
  - #2 calculated \(a_j\) in \(a_j\)'s region
Generators

- Fortress uses generator lists to express parallel iteration.
- Represented as comma-separated lists.
- Each item in the generator list can either be a boolean expression (filter) or a generator binding.
  - Generator bindings are one or more comma-separated identifiers followed by <-, then a subexpression that evaluates to an object of type \texttt{Generator}.
  - A boolean expression in a list is called a filter. A generator iteration will only be performed if the result of the filter is true.

```plaintext
for i<-1:m, j<-1:n do
  a[i,j] := b[i] c[j]
end
```
Generators

- Generators iterations should be assumed parallel unless the special sequential generator is used.
- Common generators:
  - \(1 \leq u\)
    - Range expressions
  - \(a.indexes\)
    - Index set of array
  - \(\{0, 1, 2, 3\}\)
    - Aggregate expression elements
  - \(\text{sequential}(g)\)
    - Sequential version of another generator
Generated Expressions

\begin{verbatim}
  do expr, gens end  (* #1 *)
  for gens do expr end  (* #2 *)
\end{verbatim}

• #1 is equivalent (shorthand) for #2.
The **for loop**

```
for generator do block end
```

- Parallelism is specified by the generator.
- In general, iterations should be assumed parallel unless all generators in the list are explicitly sequential.
- Each iteration is evaluated in the scope of values bound by generators.
- Body can make use of reduction variables.
Section Four

DEMOS
Task Parallelism

- An example of task parallelism: the three calls of function $f$ are executed in parallel.

```scala
println("***************************************")
println("Example of Task parallelism")
(a:ZZ32, b:ZZ32, c:ZZ32) =
  (f(1, 1, "T1"), f(2, 3, "T2"), f(5, 8, "T3"))
println("Tuple is " a " " b " " c);
```
Task Parallelism

• Here is another example, using the construct do also.

```plaintext
do
  f()
also do
  g()
also do
  h()
end
```
Data Parallelism

- Each summation is performed in parallel.

```cpp
println("****************************************")
println("Example of data parallelism")
m1:ZZ32[4, 4] = [1 2 3 4
                  5 6 7 8
                  9 10 11 12
                  13 14 15 16]
m2:ZZ32[4, 4] = [10 20 30 40
                  50 60 70 80
                  90 100 110 120
                  130 140 150 160]
for i <- 0#4 do
  for j <- 0#4 do
    println("Sum at [" i ", " j "] = " (m1[i,j] + m2[i,j]))
  end
end
```