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:

```
+   -   /   =   <   >   |   {   }
```

- Abbreviated operators:

```
   [\   \]   =/=   >=   ->   =>   |->
<|   |>   ⟦   ⟧   ≠   ≥   →   ⇒   ↦   ⟨   ⟩
```

- Short names in all caps:

OPLUS	DOT	TIMES	SQCAP	AND	OR	IN
⊕	·	×	⊓	∧	∨	∈

- Named:

Identifiers

- Regular:

`a`	`zip`	`trickOrTreat`	`foobar`
a	zip	$trickOrTreat$	$foobar$

- Formatted:

`a3`	`_a`	`a_`	`a_vec`	`_a_hat`	`a_max`	`foo_bar`
a_3	$\mathbf{a}$	a	$\vec{a}$	$\hat{\mathbf{a}}$	a_{max}	$\overline{foo}$

- Greek Letters:

`alpha`	`beta`	`GAMMA`	`DELTA`
α	β	Γ	Δ

- Unicode Names: `HEBREW_ALEF` 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.
  - $x^2 + 3y^2 = 0$

```
x^2 + 3 y^2 = 0
```

- Operator chains: `0 ≤ i < j < 100`
- Reduction syntax
  - $\mathit{factorial(n)} = \prod_{i \leftarrow 1 \ldots n} i$

```
    factorial(n) = ∏[i←1:n] il
```

Aggregate Expressions

- Set, array, maps, lists:

$\{2,\ 3,\ 5,\ 7\}$

$[\text{“France”} \rightarrow \text{“Paris”},\ \text{“Italy”} \rightarrow \text{“Rome”}]$

$\langle 0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13 \rangle$

- Set, array, maps, lists:

$\{x^2 \mid x \leftarrow \text{primes}\}$

$[x^2 \rightarrow x^3 \mid x \leftarrow \text{fibs},\ x < 1000]$

$\langle x(x+1)/2 \mid x \leftarrow 1\#100 \rangle$

- Matricies:

```
[1  0
 0  A]
```

$$\begin{pmatrix} 1 & 0 \\ 0 & A \end{pmatrix}$$

Dimension and Units

- Numeric types can be annotated with units

```
kineticEnergy(m:ℝ kg_, v:ℝ m_/s_):ℝ kg_ m_^2/s_2
    = (m v^2) / 2
```

- Common dimensions and units are provided in fortress standard library, e.g: *kg, m, s*
- Static safety checks
- Ex.:

`m_`	`kg_`	`s_`	`micro_s_`	`MW_`	`ns_`
m	kg	s	μs	MW	ns

Some Whitespace Sensitivity

- Whitespace must agree with precedence
  - Error: `a+b / c+d`
- Parentheses are sometimes required: `A+B∨C`
  - “+” and “∨” have no relative precedence.
- Fractions: `1/2 * 1/2`
- Subscripting (`a[m n]`) vs vector multiplication: (`a [m n]`)

Example Code (Fortress)

ASCII:

```
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

```
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)

```
$z = 0$
$r = x$
$\rho = r^T r$
$p = r$
do $i = 1, 25$
  $q = A p$
  $\alpha = \rho / \left(p^T q\right)$
  $z = z + \alpha p$
  $\rho_0 = \rho$
  $r = r - \alpha q$
  $\rho = r^T r$
  $\beta = \rho / \rho_0$
  $p = r + \beta p$
end
```

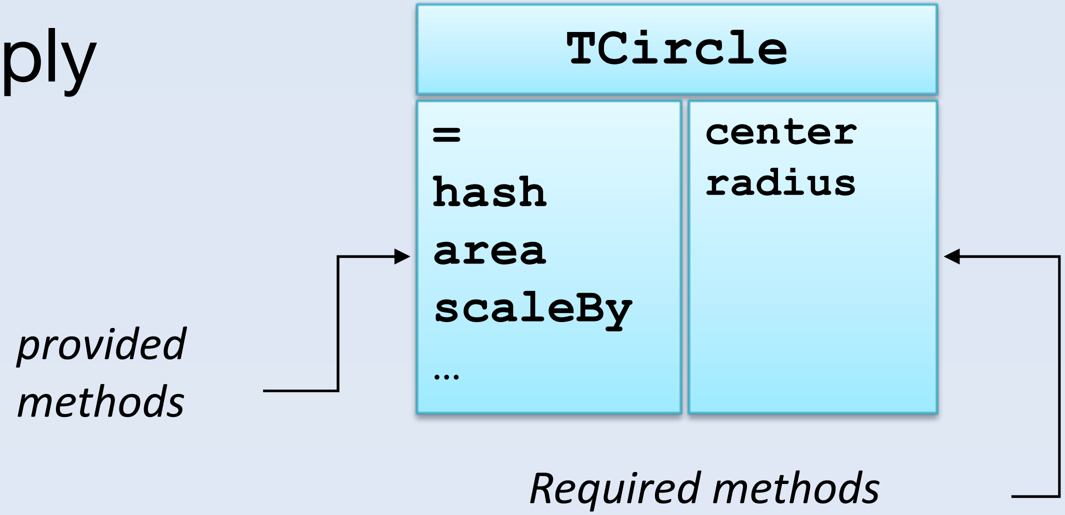
```
$z : Vec = 0$
$r : Vec = x$
$p : Vec = r$
$\rho : Elt = r^T r$
for $j \leftarrow$ seq $(1 : cgit_{\max})$ do
  $q = A p$
  $\alpha = \frac{\rho}{p^T q}$
  $z := z + \alpha \rho$
  $r := r - \alpha q$
  $\rho_0 = \rho$
  $\rho := r^T r$
  $\beta = \frac{\rho}{\rho_0}$
  $p := r + \beta p$
end
```

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.



`TCircle`

`=`
`hash`
`area`
`scaleBy`
…

`center`
`radius`

provided methods

Required methods

Examples

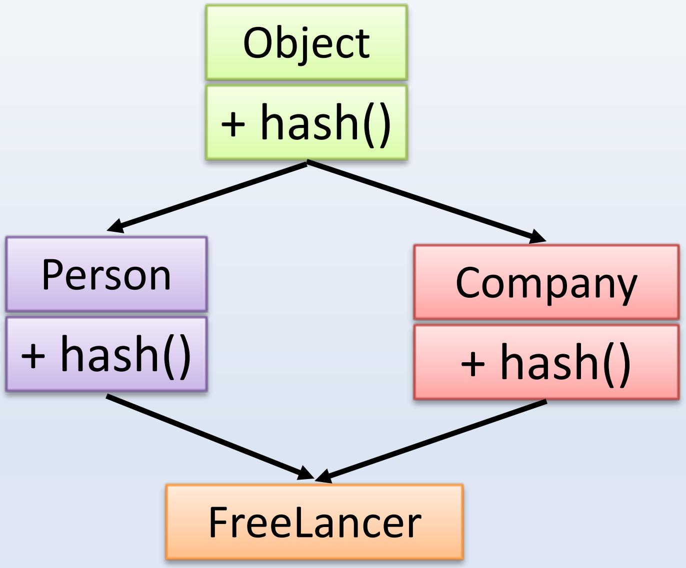
```
trait Loc
    getter position()    : (ℝ, ℝ)
    displace(nx:ℝ, ny:ℝ) : ()
end
```

```
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

```
applyN(add1, 4, 3)
(composeN(add1, 4))(3)
```

```
add1(n: ℤ): ℤ = n + 1

applyN(f: ℤ→ℤ, n: ℕ, x: ℤ): ℤ = do
   v: ℤ = x
   remaining: ℕ = n
   while remaining > 0 do
      v := f(v)
      remaining -= 1
   end
   v
end

composeN(f: ℤ→ℤ, n: ℕ): ℤ→ℤ =
   if (n = 0) then
      fn(x: ℤ) ⇒ x
   else
      base = composeN(f, n-1)
      fn(x: ℤ) ⇒ f(base(x))
   end
```

Functional Programming

- Tagged unions
- Pattern matching
- List comprehensions

```
x = ⟨ 2, 4, 6, 8, 10 ⟩

x = ⟨ x | x ← 1:10,
   iseven(x) ⟩

iseven(x: ℤ): Bool =
  x MOD 2 = 0
```

```
trait List comprises { Cons, Nil }
end

object Cons(h: ℤ, t: List) extends
List
  head: ℤ = h
  tail: List = t
end

object Nil extends List
end

sum(l: List) = typecase l of
   List ⇒ l.head + sum(l.tail)
   Nil  ⇒ 0
end
```

Operator Overloading

- Can be alphanumeric: `a 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 MINUS(m:ℤ, n:ℤ) = m - n
opr NEG(m:ℤ) = -m
opr (n:ℤ)FAC = if n ≤ 1 then 1 else n (n-1)FAC end

run(args:String...):() = do
    println(7 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.

```
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.

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

```
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

```
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:

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

- Tests consist of data plus code:

```
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

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:

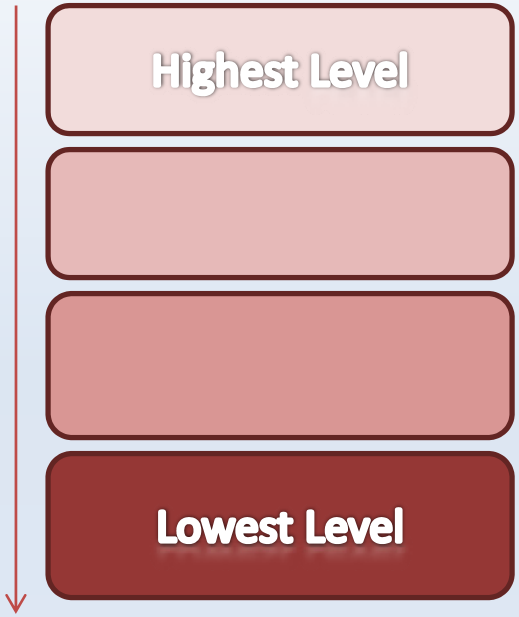
```
r:Z64:=0
(r:=1, while r=0 do end)
```

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”



Libraries to allocate locality-aware arrays

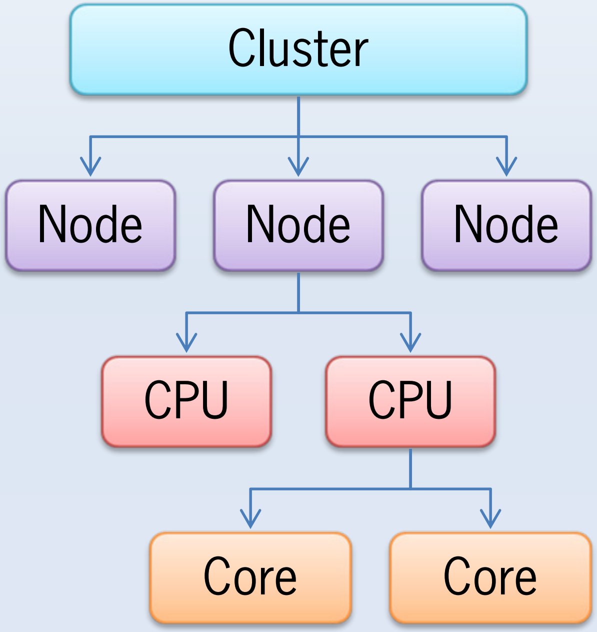
Library of Distributions

`at` Expression

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

```
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,
       at a.region(j) do
         a_j
       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 `Generator`.
    - A boolean expression in a list is called a filter. A generator iteration wil only be performed if the result of the filter is true.

```
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:
  - `l:u`
    Range expressions
  - `a.indices`
    Index set of array
  - `{0,1,2,3}`
    Aggregate expression elements
  - `sequential(g)`
    Sequential version of another generator

Generated Expressions

```
do expr, gens end                (* #1 *)
for gens do expr end             (* #2 *)
```

- #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.

```
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`.

```
do
  f()
also do
  g()
also do
  h()
end
```

Data Parallelism

- Each summation is perfomed in parallel.

```
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
```