Fortress Features and Lessons Learned

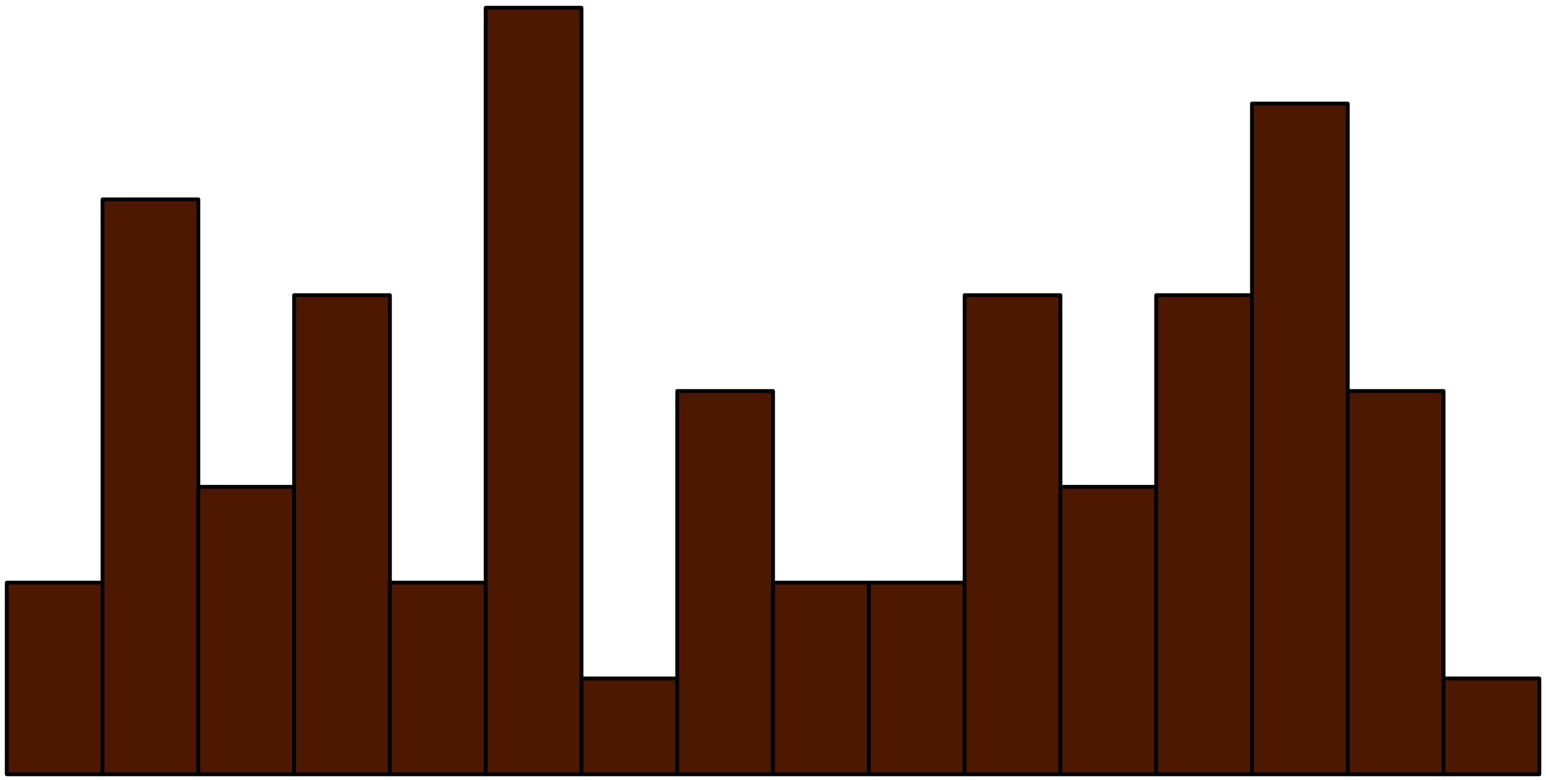
Guy L. Steele Jr.
Software Architect, Oracle Labs

Toy Programming Problem (1 of 3)

2	6	3	5	2	8	1	4	2	2	5	3	5	7	4	1

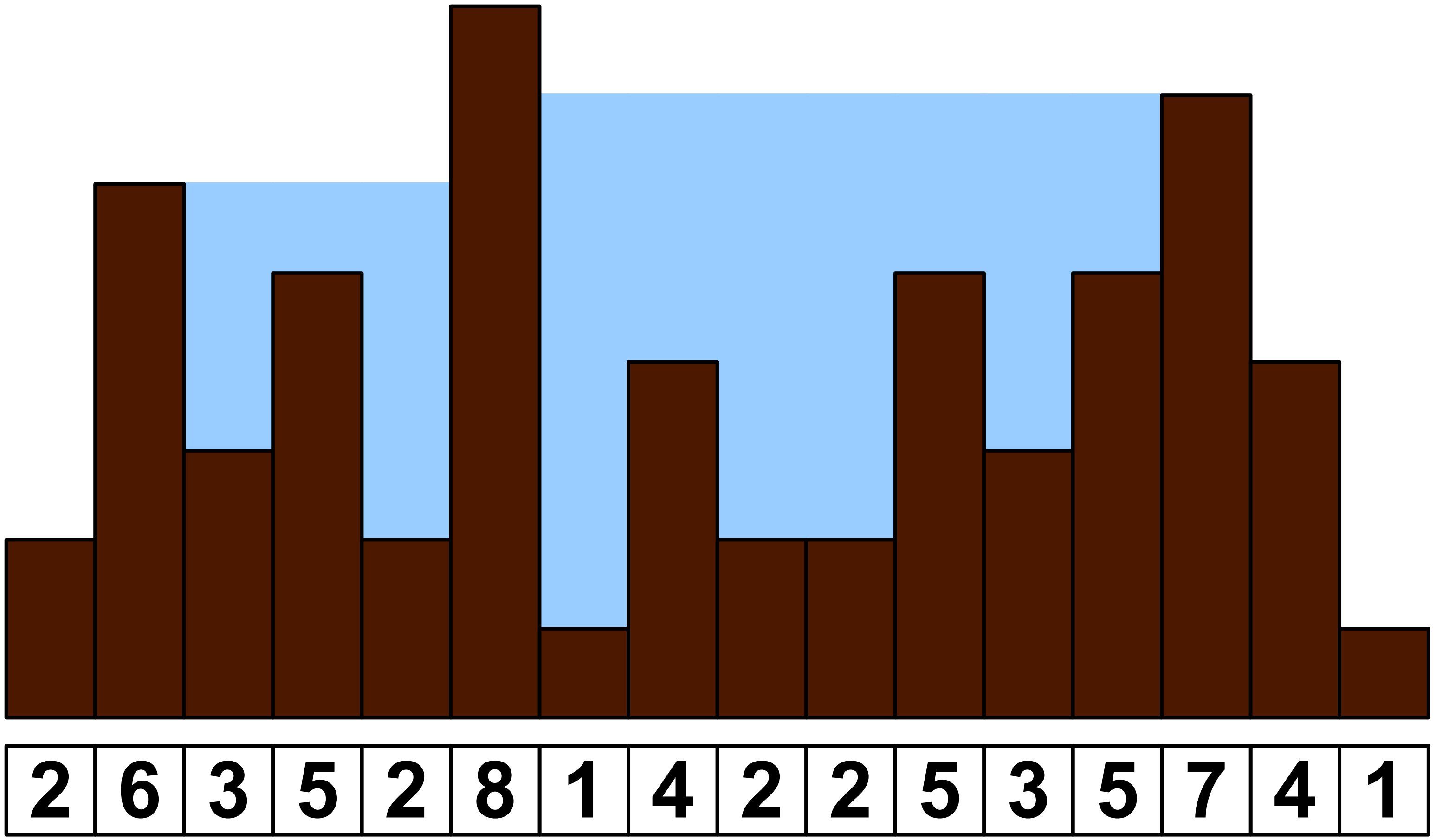
Thanks to Dan Nussbaum and Steve Heller for suggesting this toy problem.

Toy Programming Problem (2 of 3)

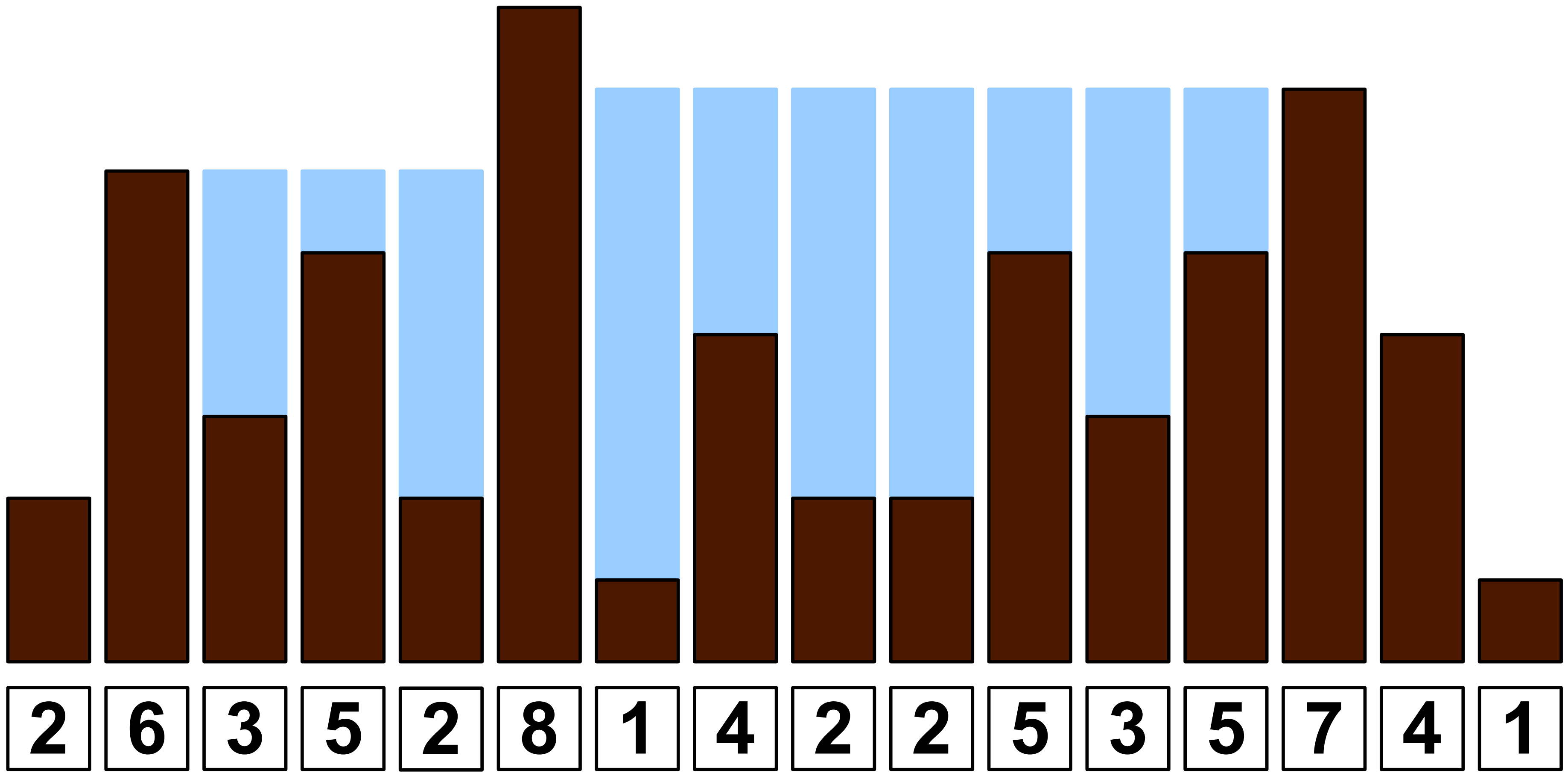


2	6	3	5	2	8	1	4	2	2	5	3	5	7	4	1

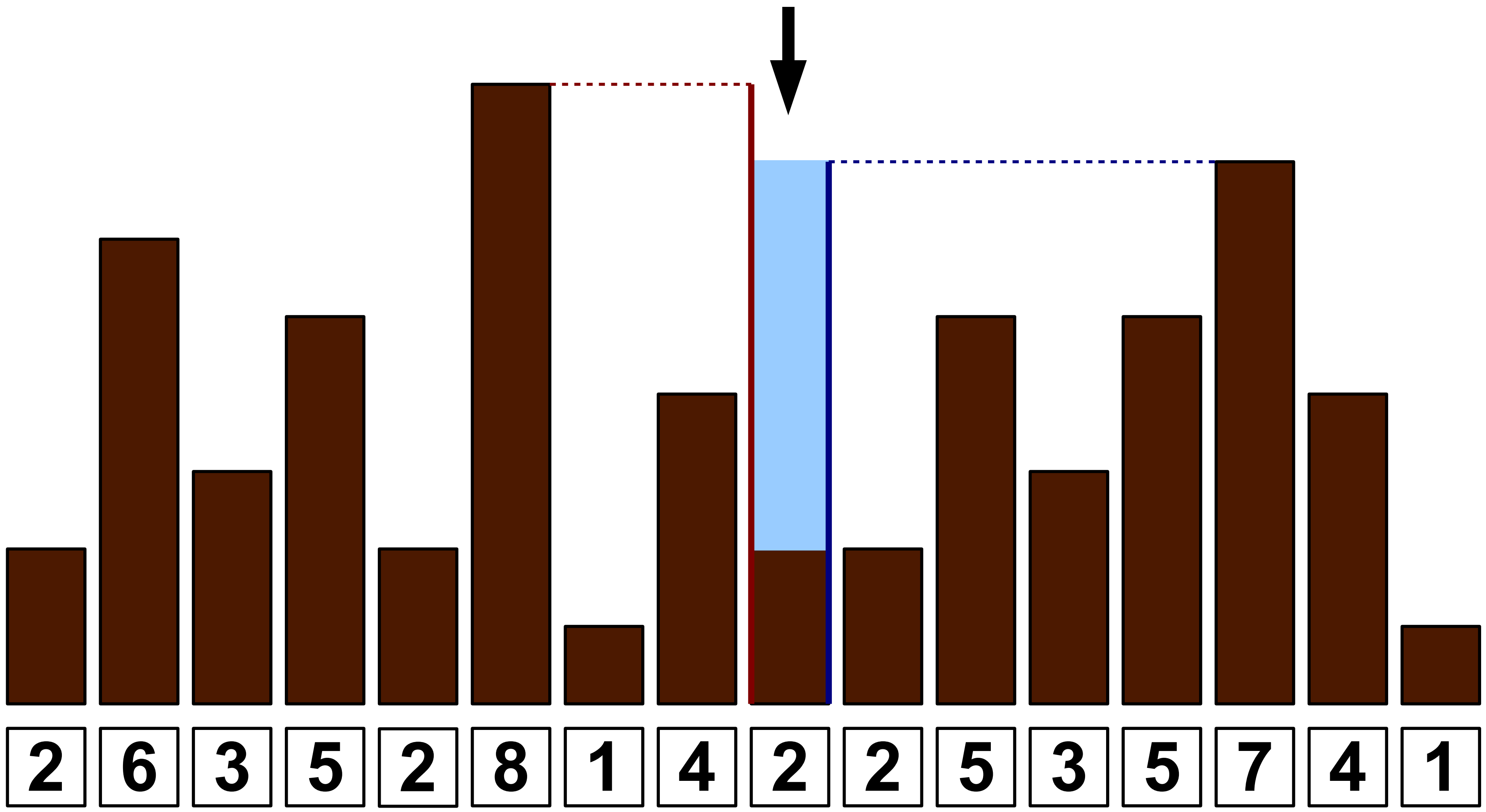
Toy Programming Problem (3 of 3)



Key Insight (1 of 2)



Key Insight (2 of 2)



Concise Solution in Fortress

$$\mathit{histogramWater}(x\colon \mathrm{List}[\![\mathbb{Z}32]\!])\colon \mathbb{Z}32 = \sum_{(v,\mathit{left},\mathit{right})\leftarrow \mathit{zip}(x,\overrightarrow{\mathtt{MAX}}\, x,\overleftarrow{\mathtt{MAX}}\, x)} ((\mathit{left}\ \mathtt{MIN}\ \mathit{right}) - v)$$

Purely functional

Allows map-reduce parallelism (more about this later)

See my Google Tech Talk
Four Solutions to a Trivial Problem on YouTube.

Three Principles for Fortress

- A growable language
- A mathematical language
- A parallel language

We tried very hard to avoid including any language feature not clearly justified by one or more of these principles.

Growability

We wanted a highly user-extensible language; therefore:

- Modular libraries
  - Components and APIs; separate compilation
  - Multiple, co-existing implementations of the same interface
- Break with Fortran's emphasis on arrays
  - Objects (with *multiple inheritance* of traits) as the substrate
    - ⋆ Why inheritance? Sharing of code.
    - ⋆ Why multiple? Even more sharing of code.
  - Support distributed/hierarchical implementations of arrays
- Therefore objects with *multimethods*
  - Type-directed dispatch on multiple arguments

Paradigmatic Examples

- Extensible numeric hierarchy
  - Integers and floats in multiple sizes (including "big")
  - Rational, complex, and quaternion
  - Interval arithmetic (to support interval solvers)
- Vectors and matrices (more generally, tensors)
  - Defining matrix-vector and vector-matrix multiplication
- Collections (lists, sets, multisets, maps)
- Arrays (multidimensional, distributed, sparse)

In all cases, we wanted to allow multiple implementations of the same interface (possibly with different performance characteristics) to coexist within the same program.

User-defined Operator Overloading

Included. Why?

- To reflect mathematical usage and tradition
  - Admittedly not itself designed with programming in mind
- Bad experience with `BigInteger` and `BigDecimal` in the Java^{TM} programming language
  - Compare Gosling's bad experience with operator overloading in C++ (that's why Java *doesn't* have it)

Growability Requires Reliability

- Strong types
  - Both types and methods may have parameters
  - Arguments may be types, operator names, integers, booleans, physical dimensions, physical units . . .
- Design-by-contract features
  - Pre- and post-conditions on functions and methods
  - Invariants on objects and traits
    - ⋆ Such invariants can be inherited
    - ⋆ We built up a rich library of algebraic traits.

$\texttt{trait}\ \mathbb{Z}\ \texttt{extends}\ \{\,\mathrm{Ring}[\![\mathbb{Z}, +, \times]\!], \mathrm{PartialOrder}[\![\mathbb{Z}, <, \leq, \geq, >]\!]\,\}$

$\quad \ldots$

$\texttt{end}$

Examples of Algebraic Traits

```
$\texttt{trait}$ BinaryOperator$\llbracket T$ $\texttt{extends}$ BinaryOperator$\llbracket T, \odot\rrbracket$, $\texttt{opr}$ $\odot\rrbracket$
  $\texttt{opr}$ $\odot(\texttt{self}, other\colon T)\colon T$
$\texttt{end}$

$\texttt{trait}$ Associative$\llbracket T$ $\texttt{extends}$ Associative$\llbracket T, \odot\rrbracket$, $\texttt{opr}$ $\odot\rrbracket$
    $\texttt{extends}$ $\{$ BinaryOperator$\llbracket T, \odot\rrbracket$, EquivalenceRelation$\llbracket T, =\rrbracket$ $\}$
  $\texttt{property}$ $\forall(a\colon T, b\colon T, c\colon T)\ \big((a \odot b) \odot c\big) = \big(a \odot (b \odot c)\big)$
$\texttt{end}$

$\texttt{trait}$ Commutative$\llbracket T$ $\texttt{extends}$ Commutative$\llbracket T, \odot\rrbracket$, $\texttt{opr}$ $\odot\rrbracket$
    $\texttt{extends}$ $\{$ BinaryOperator$\llbracket T, \odot\rrbracket$, EquivalenceRelation$\llbracket T, =\rrbracket$ $\}$
  $\texttt{property}$ $\forall(a\colon T, b\colon T)\ (a \odot b) = (b \odot a)$
$\texttt{end}$
```

Properties can be checked by unit testing. (Future work: theorem prover.)

Big Example of an Algebraic Trait

```
trait BooleanAlgebra⟦T extends BooleanAlgebra⟦T, ⋏, ⋎, ∼, ⊕, zero, one⟧,
                          opr ⋏, opr ⋎, opr ∼, opr ⊕, ident zero, ident one⟧
    extends { Commutative⟦T, ⋏⟧, Associative⟦T, ⋏⟧, Idempotent⟦T, ⋏⟧,
              Commutative⟦T, ⋎⟧, Associative⟦T, ⋎⟧, Idempotent⟦T, ⋎⟧,
              HasIdentity⟦T, ⋏, one⟧, HasIdentity⟦T, ⋎, zero⟧,
              Complements⟦T, ⋎, ∼, one⟧, Complements⟦T, ⋏, ∼, zero⟧,
              Distributive⟦T, ⋏, ⋎⟧, Distributive⟦T, ⋎, ⋏⟧,
              DeMorgan⟦T, ⋏, ⋎, ∼⟧, DeMorgan⟦T, ⋎, ⋏, ∼⟧,
              Ring⟦T, ⊕, IDENTITY, ⋏, zero, one⟧ }
  property ∀(a: T) (∼ (∼ a)) = a
  opr ⊕(self, other: T): T = (self ⋏ (∼ other)) ⋎ ((∼ self) ⋏ other)
end
```

A Fourth Principle: Symmetry

- Wherever possible, order should not matter
  - Order of declarations in a file should not matter
  - Order of import declarations should not matter
  - Order of declared parents should not matter for inheritance
  - Overload resolution should treat parameters symmetrically
  - Arguments not evaluated in any specific sequential order
- We did allow order to matter for:
  - Matching arguments to parameters
  - Tuple construction
  - Statement execution in a block

Sample code: Fortress versus Scala

Fortress code:

$$histogramWater(x\colon \mathrm{List}[\![\mathbb{Z}]\!])\colon \mathbb{Z} =$$
$$\quad walk(maxCachedTree\ x, 0, 0)$$
$$walk(x\colon \mathrm{Pair}, left\colon \mathbb{Z}, right\colon \mathbb{Z})\colon \mathbb{Z} =$$

⊛ Potentially parallel recursion

$$\quad walk(x.a, left, x.b.val\ \mathtt{MAX}\ right) + walk(x.b, left\ \mathtt{MAX}\ x.a.val, right)$$
$$walk(x\colon \mathrm{Leaf}, left\colon \mathbb{Z}, right\colon \mathbb{Z})\colon \mathbb{Z} =$$
$$\quad ((left\ \mathtt{MIN}\ right)\ \mathtt{MAX}\ x.val) - x.val$$

Scala code:

```
def histogramWater(x: List[BigInt]): BigInt =
  walk(maxCachedTree x, 0, 0)

def walk(x: Pair, left: BigInt, right: BigInt): BigInt =  // Sequential
  walk(x.a, left, x.b.val max right) + walk(x.b, left max x.a.val, right)

def walk(x: Leaf, left: BigInt, right: BigInt): BigInt =
  ((left min right) max x.val) - x.val
```

ORACLE

Typeset Fortress versus ASCII Fortress

Typeset Fortress code:

$$\begin{aligned}
&\mathit{histogramWater}\big(x\colon \mathrm{List}[\![\mathbb{Z}]\!]\big)\colon \mathbb{Z} = \\
&\quad \mathit{walk}(\mathit{maxCachedTree}\ x, 0, 0) \\
&\mathit{walk}(x\colon \mathrm{Pair}, \mathit{left}\colon \mathbb{Z}, \mathit{right}\colon \mathbb{Z})\colon \mathbb{Z} = \qquad \circledast \text{ Potentially parallel recursion} \\
&\quad \mathit{walk}(x.a, \mathit{left}, x.b.\mathit{val}\ \mathtt{MAX}\ \mathit{right}) + \mathit{walk}(x.b, \mathit{left}\ \mathtt{MAX}\ x.a.\mathit{val}, \mathit{right}) \\
&\mathit{walk}(x\colon \mathrm{Leaf}, \mathit{left}\colon \mathbb{Z}, \mathit{right}\colon \mathbb{Z})\colon \mathbb{Z} = \\
&\quad \big((\mathit{left}\ \mathtt{MIN}\ \mathit{right})\ \mathtt{MAX}\ x.\mathit{val}\big) - x.\mathit{val}
\end{aligned}$$

ASCII Fortress code:

```
histogramWater(x: List[\ZZ\]): ZZ =
  walk(maxCachedTree x, 0, 0)

walk(x: Pair, left: ZZ, right: ZZ): ZZ =  (*) Potentially parallel recursion
  walk(x.a, left, x.b.val MAX right) + walk(x.b, left MAX x.a.val, right)

walk(x: Leaf, left: ZZ, right: ZZ): ZZ =
  ((left MIN right) MAX x.val) - x.val
```

Some Similarities between Scala and Fortress

- Object-oriented with traits and multiple inheritance
- Overloaded functions and methods
- Parametric polymorphism of traits, functions, and methods
- Expression-based (no statement/expression dichotomy)
- Tuples and tuple types
- Collections libraries with heavy functional emphasis
  - Methods like `map`, `fold`, `reduce`, `filter`, `sorted`, `zip`
- Language features that enhance “growability”
- Unicode symbols may be used as operators
  - But Fortress actually uses them!
- Strong influence from Haskell

Some Differences between Scala and Fortress

- Fortress types are never erased
- Fully symmetric dynamic overload dispatch
  - Type parameters are sometimes inferred at run time
- Implicit as well as explicit paralellism
  - Tuples, binary operators have potentially parallel operands
  - Argument expressions in calls are potentially parallel
- Syntax strongly influenced by mathematics and whiteboards
  - Fortress libraries encourage use of $\cup$, $\cap$, $\subseteq$, $\in$ for sets
  - Also $|a|$, $\lfloor x \rfloor$, $\lceil x \rceil$, $p \wedge q$, $p \vee q$, $\neg q$, $p \oplus q$
  - Multiplication expressed as $a \cdot b$, $a \times b$, or just $a\,b$

A Most Important Resource

(whitespace)

Whitespace in Original FORTRAN (1 of 4)

```
      DO 20 I=1,125
20    SUM = SUM + X(I)
```

DO loop

Whitespace in Original FORTRAN (2 of 4)

```
      DO 20 I=1,125          DO loop
20    SUM = SUM + X(I)

      DO20I=1,125            also a DO loop
20    SUM=SUM+X(I)
```

Whitespace in Original FORTRAN (3 of 4)

```
      DO 20 I=1,125
20    SUM = SUM + X(I)
```

DO loop

```
      DO20I=1,125
20    SUM=SUM+X(I)
```

also a DO loop

```
      DO20I=1.125
20    SUM=SUM+X(I)
```

assignment

Whitespace in Original FORTRAN (4 of 4)

```
      DO 20 I=1,125           DO loop
20    SUM = SUM + X(I)

      DO20I=1,125             also a DO loop
20    SUM=SUM+X(I)

      DO20I=1.125             assignment
20    SUM=SUM+X(I)

      DO 20 I=1.125           also an assignment!
20    SUM = SUM + X(I)
```

Three Uses of Whitespace in Fortress

- Distinguishing multiple uses of vertical bars

  `{ |a| | a <- mylist, a | b }` $\{\,|a|\mid A \leftarrow mylist, a \bigm| b\,\}$

- Distinguishing subscripts from array constructors

  `a[i,j,k]` is a subscripting operation $a_{i,j,k}$

  `a [i,j,k]` is a reference to a and then an array constructor $[i, j, k]$

- Verifying intent of operator precedence

`a+b·c+d`	OK	`a + b · c + d`	OK
`a + b·c + d`	OK	`a+b · c+d`	rejected

Juxtaposition Is an Important Resource

- Scala: can use any name as *infix* or *postfix* operator
  - Operands and operators (method names) *alternate*:
    ```
    x = (item.x max item.y max item.z) min upperBound
    ```
  - The downside: no *prefix* operators other than `+ - ! ~`
- Fortress: juxtaposition is a user-defined overloaded operator
  - Used for function application and multiplication
    ```
    3 sin pi x - log x + 5 z^2 - 7 z + 2
    ```
    $3 \sin \pi\, x - \log\, x + 5\, z^2 - 7\, z + 2$
  - Smart string concatenation

    Compare: `"I found " + (n+1) + " errors in " + j + " files"`

    `"I found" (n+1) "errors in" j "files"`
  - “Tight” juxtaposition has higher precedence than “loose”
    ```
    "I generated" n(n-1) "ordered pairs"
    ```
  - The downside: operators must be distinguishable tokens
    ```
    x = (item.x MAX item.y MAX item.z) MIN upperBound
    ```

Nontransitive Operator Precedence in Fortress

- It's okay to write $a + b > c + d$
  - Why? Because $+$ has higher precedence than $>$.
- And it's okay to write $p < 0 \vee p > 9$
  - Why? Because $>$ has higher precedence than $\vee$.
- But in Fortress it is **not** okay to write $a + b \vee c + d$.
  - There is no precedence defined between $+$ and $\vee$.
- We use only the most obvious and familiar rules of precedence.
  - This reduces opportunities for for making silly mistakes.
- This matters when you have hundreds of operator symbols such as $\cup \ \cap \ \sqsubset \ \div \ \oplus \ \precsim \ \boxplus \ \otimes \ \curlywedge \ \circledcirc \ \in \ \diamond \ \star \ \Subset \ \lessdot \ \approx \ \| \ \oslash$.
- Not a big burden—simply use parentheses to make meaning clear: $(a + b) \vee (c + d)$ or $a + (b \vee c) + d$

Assignment versus Binding in Fortress

- Design decisions:
  - Make declaration easier than assignment
  - Make "final" variables easier to declare than mutable variables

    $x = h + 1$ We see this on whiteboards!

    $y\colon \mathbb{Z} := h + 1$ Declaration of mutable variable needs a type

- Assignment is ":=" rather than "="

  $y := y + 1$

- And Fortress does support "compound assignment":

  $y += 1$

- We want to encourage a "mostly pure, SSA" style of coding. (Why? It's good for you—and for parallelism.)
- But we keep the "tax" small (one character)

Comments

- Scaladoc and Fortress both support wiki syntax in comments
- Fortress wiki syntax is a slight superset of Wiki Creole 1.0:
  `//italic//   **boldface**   [[link]]   {{image}}`
  More verbose than other wiki syntaxes, but it's "standard."
- But *single* backquotes surround embedded Fortress code:
  ```
  (*) The **best** way to swap ‘x‘ and ‘y‘ is ‘(x,y) := (y,x)‘
  ```
  ⊛ The **best** way to swap x and y is $(x, y) := (y, x)$
- Frequently used notations should be clear, memorable, and concise

The Value of Comprehensions

- Fortress adopts Haskell constructor/comprehension notation
- Arrays, lists, sets, *and* multisets

```
  [1,2,3]      [ |a| | a <- mylist, a | b ]
<|1,2,3|>     <| |a| | a <- mylist, a | b |>
  {1,2,3}      { |a| | a <- mylist, a | b }
{|1,2,3|}     {| |a| | a <- mylist, a | b |}
```

$$[\,1,2,3\,] \qquad \left[\,|a| \;\middle|\; a \leftarrow mylist, a \mid b\,\right]$$

$$\langle\,1,2,3\,\rangle \qquad \left\langle\,|a| \;\middle|\; a \leftarrow mylist, a \mid b\,\right\rangle$$

$$\{\,1,2,3\,\} \qquad \left\{\,|a| \;\middle|\; a \leftarrow mylist, a \mid b\,\right\}$$

$$\{\!|\,1,2,3\,|\!\} \qquad \{\!|\,|a| \;\middle|\; a \leftarrow mylist, a \mid b\,|\!\}$$

$$\wr\,1,2,3\,\int \qquad \wr\,|a| \;\middle|\; a \leftarrow mylist, a \mid b\,\int$$

Big Reduction Operators

- What happens when you cross fold with comprehensions?

```
BIG SUM [p <- mylist, prime p] p+1
BIG MAX [k <- 1:n] a[k]
BIG OPLUS [j <- 1:n, k <- j:n] f(j,k)
```

$$\sum_{\substack{p \leftarrow mylist \\ prime\, p}} p + 1 \qquad \underset{k \leftarrow 1:n}{\mathtt{MAX}}\ a_k \qquad \bigoplus_{\substack{j \leftarrow 1:n \\ k \leftarrow j:n}} f(j,k)$$

Reduction operators are exactly like comprehensions.

Typically, reductions build numbers rather than aggregates.

Implementationally they are identical.

Language Mechanics for Parallelism (1 of 2)

- Very general big operator and comprehension syntax

  $\sum_{i \leftarrow 0:9} a_i\, x^i \quad \underset{i \leftarrow 0:9}{\mathtt{MAX}} (a_i + b_i) \quad \{ (m,n) \mid m \leftarrow A, \mathit{prime}\, m, n \leftarrow B, m \mid n \}$

  - $\sum_{i \leftarrow 0:9} a_i\, x^i$ desugars to $\sum(0:9, \mathtt{fn}\ i \Rightarrow a_i\, x^i)$

```
opr $\sum[\![E, T$ extends $\mathrm{Monoid}[\![T,+]\!]]\!]\big(g\colon \mathrm{Generator}[\![E]\!], \mathit{body}\colon E \to T\big)\colon T =$
    $g.\mathit{generate}\big(\mathrm{SumReduction}[\![T]\!], \mathit{body}\big)$

object $\mathrm{SumReduction}[\![T$ extends $\mathrm{Monoid}[\![T,+]\!]]\!]$ extends $\mathrm{Reduction}[\![T]\!]$
    $\mathit{empty}()\colon T = \mathit{cast}[\![T]\!]\, 0$
    $\mathit{join}(a\colon T, b\colon T)\colon T = a + b$
end
```

- Definition of $\sum$ can be overloaded

Language Mechanics for Parallelism (2 of 2)

- Use of generic types
  - Generators and reducers are generic in element type
  - Their methods have further generic type parameters
  - Can be (lazily) composed: cross product, catenation, etc.
  - Encode algebraic properties of each in the type system
- Generator controls sequential/parallel strategy
- Overloaded method dispatch selects best reducer implementation for a given generator
- All defined by libraries code in Fortress, not the compiler

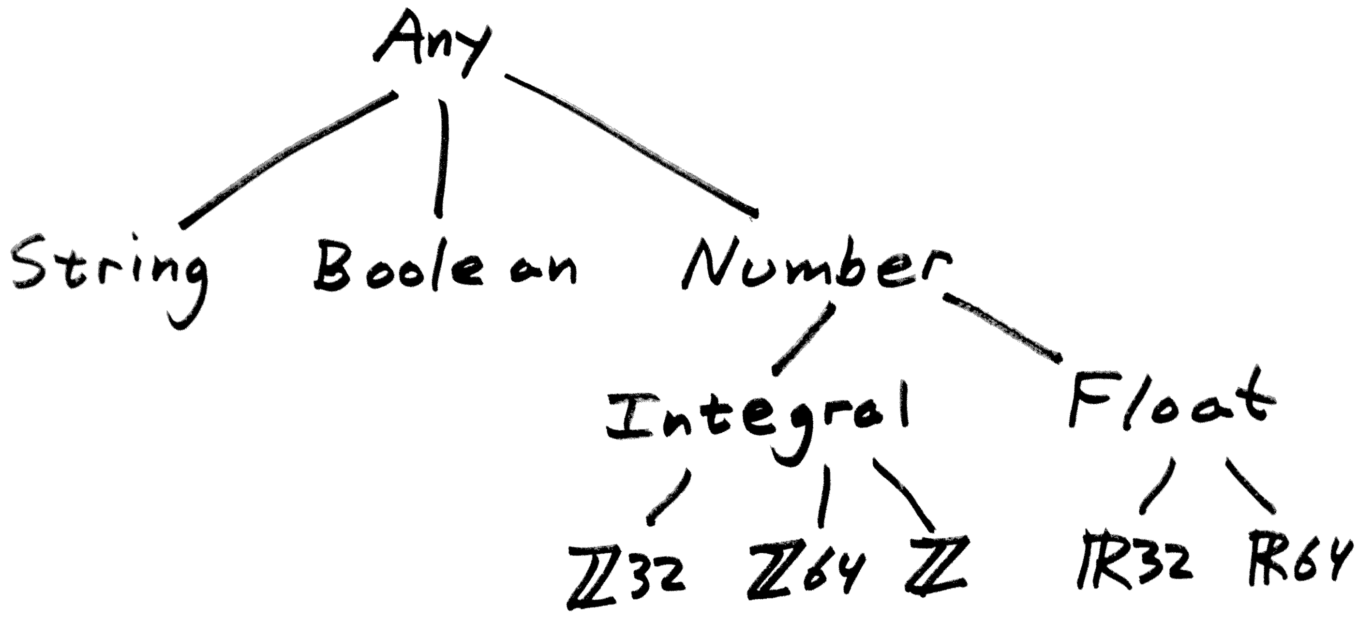
Low-level Cooperative Parallelism

- Tuples, arguments, and binary operators
  - The real basis for cooperative parallelism in Fortress:
    $walk(x.a, left, x.b.val\ \mathtt{MAX}\ right) + walk(x.b, left\ \mathtt{MAX}\ x.a.val, right)$
- Compiler figures out which subexpressions are worth making into microtasks (still an area of research)
- Microtasks automatically load-balanced through work-stealing
- Goal: do for processors what GC does for memory

Competitive Parallelism

- Fortress has a global shared memory model
- How to compete for resources, with low overhead?
  - Locks are totally pessimistic
  - Transactional memory has optimistic strategies
- Fortress provides atomic blocks as an abstraction
  - `atomic do ... end` appears to execute “all at once”
  - Currently implemented using *transactional memory*
- Nesting with each other *and* exceptions is “interesting”

Type Hierarchy with Multiple Inheritance (1 of 3)



Will method overloading work?

$$print(x\colon \text{String}) = \ldots$$

$$print(x\colon \text{Boolean}) = \ldots$$

$$print(x\colon \mathbb{Z}32) = \ldots$$

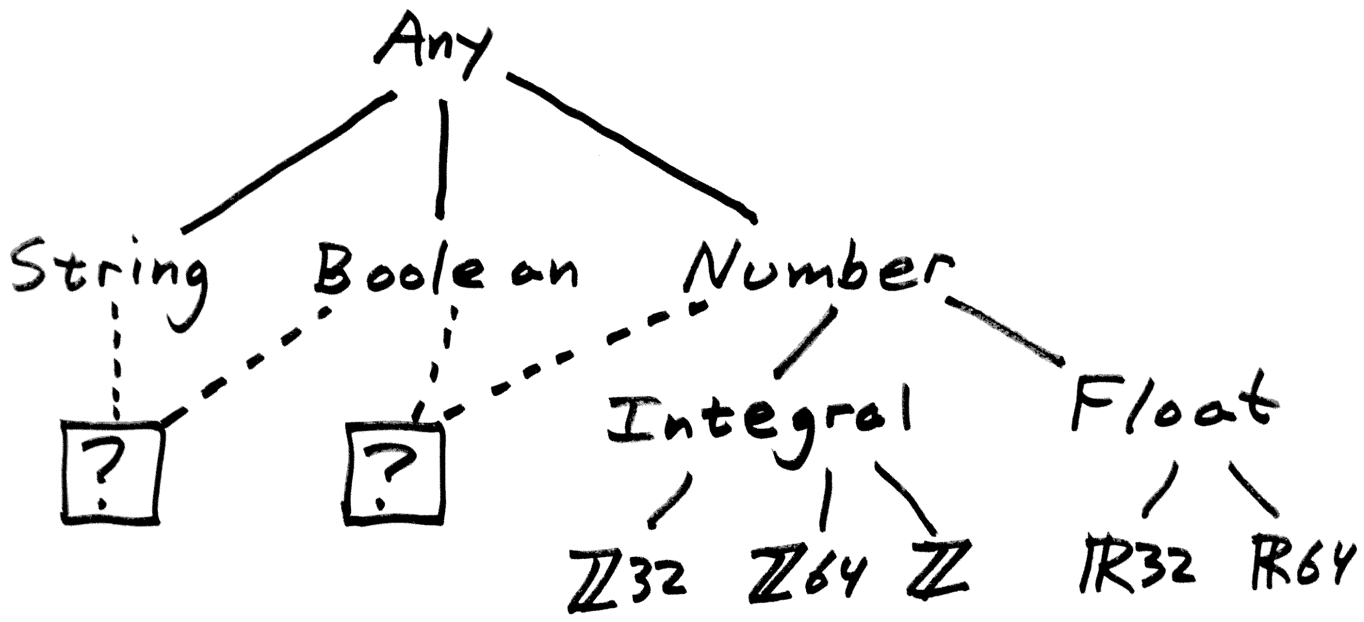
$$print(x\colon \mathbb{Z}64) = \ldots$$

$$print(x\colon \mathbb{Z}) = \ldots$$

$$print(x\colon \mathbb{R}32) = \ldots$$

$$print(x\colon \mathbb{R}64) = \ldots$$

Type Hierarchy with Multiple Inheritance (2 of 3)



How do you say you *don't* want sharing?

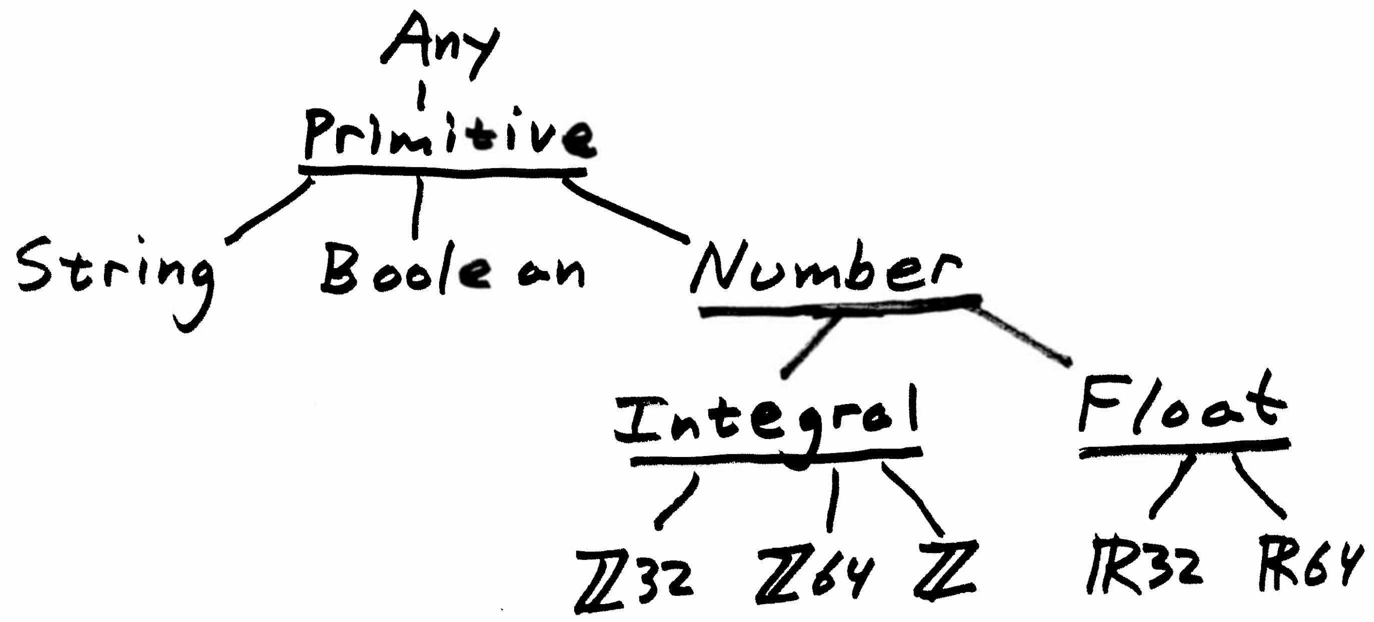
$\texttt{trait}\ \mathrm{String}\ \texttt{extends}\ \{\,\mathrm{Any}\,\}\ldots\ \texttt{end}$

$\texttt{trait}\ \mathrm{Boolean}\ \texttt{extends}\ \{\,\mathrm{Any}\,\}\ \texttt{excludes}\ \{\,\mathrm{String}\,\}\ldots\ \texttt{end}$

$\texttt{trait}\ \mathrm{Number}\ \texttt{extends}\ \{\,\mathrm{Any}\,\}\ \texttt{excludes}\ \{\,\mathrm{String}, \mathrm{Boolean}\,\}\ldots\ \texttt{end}$

Quadratic blow-up!

Type Hierarchy with Multiple Inheritance (3 of 3)



$\texttt{partitioned trait}\ \text{Primitive}\ \texttt{extends}\ \{\ \text{Any}\ \}\ \texttt{end}$

$\texttt{trait}\ \text{String}\ \texttt{extends}\ \{\ \text{Primitive}\ \}\ \ldots\ \texttt{end}$

$\texttt{trait}\ \text{Boolean}\ \texttt{extends}\ \{\ \text{Primitive}\ \}\ \ldots\ \texttt{end}$

$\texttt{partitioned trait}\ \text{Number}\ \texttt{extends}\ \{\ \text{Primitive}\ \}\ \ldots\ \texttt{end}$

Fortress Method Dispatch Mechanism

- Fully symmetric dynamic multimethod dispatch uses the “run-times types” or “actual classes” of *all* arguments
  - No need for visitor patterns (needed in single-dispatch languages like Java)
- Types are never erased
  - Dynamic dispatch can distinguish $\mathrm{List}[\![\mathrm{Boolean}]\!]$ from $\mathrm{List}[\![\mathrm{String}]\!]$
  - Both are dynamically more specific than $\mathrm{List}[\![T]\!]$

$$crunch(x\colon \mathrm{List}[\![\mathrm{Boolean}]\!]) = \ldots$$

$$crunch(x\colon \mathrm{List}[\![\mathrm{String}]\!]) = \ldots$$

$$crunch[\![T]\!](x\colon \mathrm{List}[\![T]\!]) = \ldots$$

$$myList\colon \mathrm{List}[\![\mathrm{Object}]\!] = \ldots$$

$crunch(myList)$ could end up invoking any of the three definitions

Fortress Run-Time Type Inference (1 of 2)

- In the general case, must compute values for type parameters such as T based on the “run -time types” of the argument values

$$crunch(x\colon \mathrm{List}[\![\mathrm{Boolean}]\!]) = \ldots$$

$$crunch(x\colon \mathrm{List}[\![\mathrm{String}]\!]) = \ldots$$

$$crunch[\![T]\!](x\colon \mathrm{List}[\![T]\!]) = \ldots$$

$$myList\colon \mathrm{List}[\![\mathrm{Object}]\!] = \text{a list of threads}$$

$$crunch(myList)$$

needs to infer at run time that $T = \mathrm{Thread}$

Fortress Run-Time Type Inference (2 of 2)

- Methods may have multiple static parameters
- Each static parameters may have declared bounds
- The bounds may depend on all the static parameters

$$\begin{aligned}
&\mathit{mangle}\Big[\!\!\Big[M\ \texttt{extends}\ \big\{\, \mathrm{PartialOrder}[\![M]\!], \mathrm{Group}[\![M,+]\!] \,\big\}, \\
&\qquad\quad F\ \texttt{extends}\ M \to N, \\
&\qquad\quad N\ \texttt{extends}\ \mathrm{List}[\![M]\!] \Big]\!\!\Big] \\
&\qquad (x\colon M, f\colon F)\colon N = \ldots
\end{aligned}$$

- In general, this requires solving a system of type inequalities at run time.

Where We Got Stuck

- Interaction of generic methods with symmetry requirement
- Symmetry requirements led to general parametric formulations of the meet rule and the return type rule
- It turns out this requires nontrivial constraint solving at run time
  - “Non-trivial” is a euphemism for “exponential cost”
  - Contravariance and union types interact badly
- We limited this by imposing a lexical asymmetry on type parameters of methods
- But another nasty case popped up: the return type rule has nontrivial consequences

So we had a grand vision, but could not quite pull it off.

We Learned Some Good Things

- Implementing generic types with multiple inheritance of generic methods and symmetric dynamic dispatch
- Parallelism with automatic work-stealing
- Factoring parallelism on collections using generators and reducers
- Parsing a more mathematical syntax
- Pretty-printing mathematical syntax
- Nontransitive operator precedence
- Implementing checking of physical dimensions and units in the context of a generic type system

My current thinking is to back off and try to exploit a subset of these ideas using a less complex type system.

Another Point about Syntax

- There are tradeoffs between readability and writability
- Mathematical syntax was designed more for certain kinds of concision than for robustness!
  - In particular, simple juxtaposition is used in many ways
  - We tried to make this programmable
    - ⋆ Sometimes this worked beautifully
    - ⋆ Sometimes it required some weird contortions
- Haskell and Scala do this more consistently
  - Each captures a certain PART of mathematical tradition quite well

Reflections on Trying

I knew when I started the Fortress project in 2003 that having it take market share from either Fortran or Java was a long shot.

On the other hand, I wanted to assume that it would be wildly successful, and we proceeded accordingly.

We ran into several hard problems, solved some of them, and revised the design over the years.

Did we revise too much? Well, I think we are too often fearful of correcting mistakes.

(The early development of JavaScript was a bit too cautious. Then again, maybe Worse Is Better.)

Programming Language Life Cycles?

- Most languages don’t thrive forever
- OTOH, many languages don’t die completely
- Good ideas survive by hopping from one language to another
  - The language that develops an idea may not be the one that survives
  - Instead, a “worse is better” language may popularize it
- Sometimes a new combination of old ideas is the best new idea
  - I believe in studying history
- Maybe a few ideas from Fortress will prove useful
  - Swift allows binding in `if` statements, much as in Fortress:
    $\texttt{if}\ x \leftarrow z\ \texttt{then}\ f(x)\ \texttt{else}\ y\ \texttt{end}$
    where the type of z is $\mathrm{Maybe}[\![T]\!]$

ORACLE

Greenspun’s Tenth Rule (circa 1993)

Any sufficiently complicated C or Fortran program contains an ad hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp.

Greenspun’s Tenth Rule (circa 1993)

> Any sufficiently complicated C or Fortran program contains an ad hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp.

Here’s a modern variation:

> Any sufficiently complicated parallel program contains an ad hoc, informally-specified, bug-ridden, slow implementation of . . . what?

We need to figure out what “what” is.

ORACLE®