

Formalism in the Fortress Programming Language

Sukyoung Ryu

Sun Microsystems Laboratories

April 11, 2006

Outline

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

Fortress: “To Do for Fortran What Java™ Did for C”

- Catch “stupid mistakes”
- Extensive libraries
- Platform independence
- Security model, including type safety
- Multithreading
- Dynamic compilation

- Make programmers more productive

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**

Key Ideas

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

Outline

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

Minimalist Approach

- As few primitive types as possible (cf. Bacon's Kava)
- User-defined parameterized types
- User-defined polymorphic operators
- Aggressive type inference to reduce clutter
- Aggressive static and dynamic optimization
- Complex, Rational, Interval, Vector, Matrix, like Hashtable, are defined by library, coded in Fortress

- NOTE: we want nice notation, not `x.multiply(y)`
- These are existing techniques—let's put 'em to work

Interesting Language Design Strategy

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

A Growable, Open Language

- Old language design model:
 - > Study applications
 - > Add language features to improve application coding
- Our new model:
 - > Study applications
 - > Study how a library can improve application coding
 - > Add language features to improve library coding
- Conjectures:
 - > Better leverage, leading to more rapid improvement
 - > Enables experimentation with open-source strategies

Replaceable Components

- Avoid a monolithic “Standard Library”
- Replaceable components with version control
- Encourage alternate implementations
 - > Performance choices
 - > Test them against each other
- Encourage experimentation
 - > Framework for alternate language designs

Type System: Objects and Traits

- Traits: like interfaces, but may contain code
 - > Based on work by Schärli, Ducasse, Nierstrasz, Black, et al.
- Multiple inheritance of code (but not fields)
 - > Objects with fields are the leaves of the hierarchy
- Multiple inheritance of contracts and tests
 - > Automated unit testing
- Traits and methods may be parameterized
 - > Parameters may be types or compile-time constants
- Primitive types are first-class
 - > Booleans, integers, floats, characters are all objects

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  $\forall(a: T, b: T) (a \sim b) \leftrightarrow (b \sim a)$ 
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.)

Outline

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

Conventional Mathematical Notation

- The language of mathematics is centuries old, concise, convenient, and widely taught
- Programming language notation can become closer to mathematical notation (Unicode helps a lot)
 - > $\mathbf{v_norm} = \mathbf{v} / \|\mathbf{v}\|$
 - > $\mathbf{\Sigma[k=1:n] a[k] x^k}$
 - > $\mathbf{C = A \cup B}$
 - > $\mathbf{y = 3 x \sin x \cos 2 x \log \log x}$
- Parsing this stuff is an interesting research problem
- We see benefits in using notations for programming that are also used for specification

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
- Juxtaposition is a binary operator
- Any other operator can be infix, prefix, and/or postfix
- Many sets of brackets
- Conservative, traditional rules of precedence
 - > A dag, not always transitive (examples: $A+B>C$ is okay; so is $B>C \vee D>E$; but $A+B \vee C$ needs parentheses)

Libraries Define . . .

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

```
opr -(m:Integer,n:Integer) = m.subtract(n)
```

```
opr -(m:Integer) = m.negate()
```

```
opr (n:Integer)! = if n=0 then 1 else n*(n-1)! end
```

- Whether a juxtaposition is meaningful

```
opr juxta(m:Integer,n:Integer) = m.times(n)
```

- What bracketing operators actually mean

```
opr [x:Number] = ceiling(x)
```

```
opr |x:Number| = 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.

The form $x:T=e$ declares a variable x of type T with initial value e, and that variable may be updated using the assignment operator $:=$.

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, EIt)
  cgit_max = 25
  z: Vec = 0
  r: Vec = x
  p: Vec = r
  rho: EIt = 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||)

```

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

The form $x=e$ as a statement declares variable x to have an unchanging value. The type of x is exactly the type of the expression e .

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

Simple Example: NAS CG Kernel (Unicode)

```

conjGrad[[Elt extends Number, nat N,
         Mat extends Matrix[[Elt,N×N]],
         Vec extends Vector[[Elt,N]]
        ]](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 abbreviated 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

```

conjGrad [[Elt extends Number, nat N,
           Mat extends Matrix [[Elt,  $N \times N$ ]],
           Vec extends Vector [[Elt, N]]
          ]](A:Mat, x:Vec):(Vec, Elt)

```

```
cgmax = 25
```

```
z:Vec = 0
```

```
r:Vec = x
```

```
p:Vec = r
```

```
 $\rho$ :Elt =  $r^T r$ 
```

```
for j ← seq(1:cgmax) do
```

```
  q = A p
```

```
   $\alpha = \frac{\rho}{p^T q}$ 
```

```
  z := z +  $\alpha$  p
```

```
  r := r -  $\alpha$  q
```

```
   $\rho_0 = \rho$ 
```

```
   $\rho := r^T r$ 
```

```
   $\beta = \frac{\rho}{\rho_0}$ 
```

```
  p := r +  $\beta$  p
```

```
end
```

```
(z, ||x - Az||)
```

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
ρ = rT r
p = r
DO i = 1, 25
    q = A p
    α = ρ / (pT q)
    z = z + α p
    ρ0 = ρ
    r = r - α q
    ρ = rT r
    β = ρ / ρ0
    p = r + β p
ENDDO
compute residual norm explicitly: ||r|| = ||x - A z||
  
```

```

z: Vec = 0
r: Vec = x
p: Vec = r
ρ: Elt = rT r
for j ← seq(1:cgitmax) do
    q = A p
    α =  $\frac{\rho}{p^T q}$ 
    z := z + α p
    r := r - α q
    ρ0 = ρ
    ρ := rT r
    β =  $\frac{\rho}{\rho_0}$ 
    p := r + β 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

- The Fortress Programming Language
 - > Growing a Language
 - > Mathematical Notation
 - > **Parallelism by Default**
- Formalism in 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. (This is not a property of machines, but of people.)

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
 - > Maybe you should have to ask for sequential execution!
- Fortress “loops” are parallel by default
 - > This is actually a library convention about generators

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,j)←a.indices 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.rowMajor do
  print a[i,j]
end
```

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

Generators

- Aggregates
 - > Lists $\langle 1, 2, 4, 3, 4 \rangle$ and vectors $[1\ 2\ 4\ 3\ 4]$
 - > Sets $\{1, 2, 3, 4\}$ and multisets $\{|1, 2, 3, 4, 4|\}$
 - > Arrays (including multidimensional)
- Ranges $1:10$ and $1:99:2$
- Index sets $a.indices$ and $a.indices.rowMajor$
- Index-value sets $ht.keyValuePairs$

Loops, Reducers, Comprehensions

for $k \leftarrow 1:n$ **do** **print** i **end**

$y = \Sigma[k \leftarrow 1:n] a[k] x^k$

$z = \Sigma S$ (* same as $\Sigma[x \leftarrow S] x$ *)

$v = \cap[k \leftarrow S, \text{prime } k] \text{arrayOfSets}[k]$

$w = \text{MAX}[(i,j) \leftarrow a.\text{indices}] a[i,j]$

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

$\langle x^2 \mid x \leftarrow 1:100 \rangle$

Regions and Distributions

- Regions describe CPU and memory resources and their properties
 - > Allocation heaps
 - > Parallelism
 - > Memory coherence
- Distributions describe how to map aggregates onto regions.
 - > Block, block-cyclic, etc., and user-definable!
 - > Map an array into a chip? Use a local heap.
 - > Map an array onto a cluster? Break it up.

Outline

- The Fortress Programming Language
- Formalism in Fortress
 - > Formalizing Language Semantics
 - > Types Example: “where” Clauses
 - > Types Example: Arrays


Formalism for the Fortress Programming Language

Eric Allen
Eric.Allen@sun.com

Sukeyoung Ryu
Sukeyoung.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

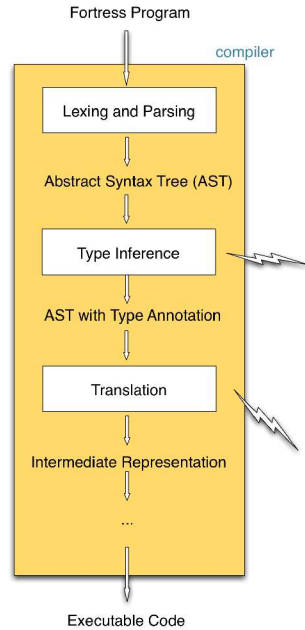
Accumulated rounding error in patriot missile software caused a missile to track its target incorrectly.

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

- Allows proofs of soundness and formal analysis



Typing Rules

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

[T-VAR] $\frac{}{p; \Delta, \Gamma \vdash x : \Gamma(x)}$

[T-SELF] $\frac{}{p; \Delta, \Gamma \vdash \text{self} : \Gamma(\text{self})}$

[T-OBJECT] $\frac{\text{object } O : \langle \dots, \tau_i \rangle, e \in p \quad p; \Delta \vdash O(\tau_i) \text{ ok} \quad p; \Delta, \Gamma \vdash \tau_i \leq \tau \quad p; \Delta \vdash \tau \leq \tau'}{p; \Delta, \Gamma \vdash O(\tau_i)(e) : O(\tau)}$

[T-FIELD] $\frac{p; \Delta, \Gamma \vdash e_1 : \tau_1 \quad \text{bound}_d(\tau_2) = O(\tau_1) \quad \text{object } O(\alpha \leq \tau_1) \text{ ok} \quad \tau_2 \leq \tau'}{p; \Delta, \Gamma \vdash e_1.e_2 : \tau_2}$

[T-METHOD] $\frac{p; \Delta, \Gamma \vdash e_1 : \tau_1 \quad \text{mtype}_d(f, \text{bound}_d(\tau_2)) = \{(\alpha \leq \tau_1), \tau' \rightarrow \tau'_2\} \quad p; \Delta \vdash \tau \text{ ok} \quad p; \Delta \vdash \tau' \leq \tau_1 \leq \tau \quad p; \Delta, \Gamma \vdash \tau' \leq \tau'_2 \quad p; \Delta, \Gamma \vdash e_2 : \tau'_2}{p; \Delta, \Gamma \vdash e_1.f(\tau_2) : \tau'_2}$

Evaluation Rules

Evaluation rules: $p \vdash E(R) \rightarrow E(L)$

[R-FIELD] $\frac{\text{object } O(\alpha \leq \tau_1) \text{ ok} \quad \tau_2 \leq \tau' \quad e_1 \in p \quad p \vdash E(O(\tau_1)(\tau_2).e_2) \rightarrow E[\tau_1/\tau_2]e_1}{p \vdash E(e_1.e_2) \rightarrow E[\tau_1/\tau_2]e_2}$

[R-METHOD] $\frac{\text{object } O : \langle \tau_i, \dots \rangle \in p \quad \text{mbody } f(\tau_1, \dots, O(\tau_1)) = \langle \tau' \rangle \rightarrow e}{p \vdash E(O(\tau_1)(\tau_2).f(\tau_2)) \rightarrow E[\tau_1/\tau_2]O(\tau_1)/\text{self}[\tau_2/\tau_1]e}$

Type Soundness Proof

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

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

Case [R-FIELD]: $e = E[\tau_1/\tau_2]e_2$

By the well-typedness of e , we have $p; \Delta, \Gamma \vdash O(\tau_1)(\tau_2).e_2 : \tau_2$ where $\text{object } O(\alpha \leq \tau_1) \text{ ok} \quad \tau_2 \leq \tau'$, $\tau_2 \leq \tau_1$, and $e_2 \in p$.

By typing rules [T-OBJECT], [T-OBJECTDEF], [T-FIELDDEF], and [W-BOTH], we have:

- (1a) $p; \Delta, \Gamma \vdash \tau_1 \leq \tau'$
- (1b) $p; \Delta, \Gamma \vdash \tau_2 \leq \tau'$
- (2a) $p; \Delta, \alpha \leq \tau_1, \tau_2 \vdash e_2 : \tau''$
- (2b) $p; \Delta, \alpha \leq \tau_1 \vdash \tau'' \leq \tau''$
- (3b) $p; \Delta, \Gamma \vdash \tau' \leq \tau_2 \leq \tau'$
- (4a) $p; \Delta, \Gamma \vdash O(\tau_1)(\tau_2) : O(\tau_2)$

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

- (5a) $p; \Delta, \alpha \leq \tau_1, \tau_2 \vdash [e_2/\tau_2]e_2 : \tau''$
- (5b) $p; \Delta, \alpha \leq \tau_1 \vdash \tau'' \leq \tau''$

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

- (6a) $p; \Delta, \tau_1/\tau_2 \vdash [e_2/\tau_2]e_2 : \tau''$
- (6b) $p; \Delta, \tau_1/\tau_2 \leq \tau_2$

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

- (7a) $p; \Delta \vdash \tau'' \leq \tau_2$
- (8a) $p; \Delta \vdash \tau'' \leq \tau'$

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

Case [R-METHOD]: ... □

Example Program in Fortress

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

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

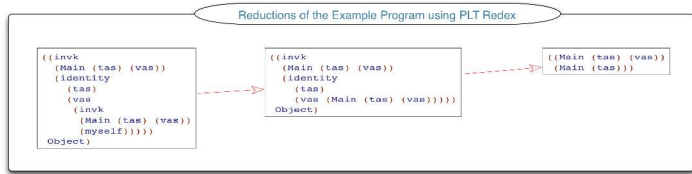
Mechanized Semantics

- Tests soundness of language semantics

Soundness of the Example Program

Suppose p is the example program.

If $p; \emptyset, \emptyset \vdash \text{Main}[]() \text{.identity}[](\text{Main}[]() \text{.myself}) : \text{Object}$ and $p \vdash \text{Main}[]() \text{.identity}[](\text{Main}[]() \text{.myself}) \rightarrow \text{Main}[]()$ then $p; \emptyset, \emptyset \vdash \text{Main}[]() : \text{Main}[]()$ where $p; \emptyset \vdash \text{Main}[]() \leq \text{Object}$.



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)

Mechanizing Language Semantics

- Testing vs Proving
- The POPLmark Challenge
 - > ACL2, ATS, Coq, HOL, HOL Light, Isabelle, MetaPRL, PVS, Twelf
- Mechanized Soundness Proofs of Featherweight Java
 - > Coq, Twelf, Isabelle/HOL
- Jinja
 - > Formalizing the source language, virtual machine, and compiler in Isabelle/HOL

Outline

- The Fortress Programming Language
- Formalism in Fortress
 - > Formalizing Language Semantics
 - > Types Example: “where” Clauses
 - > Types Example: Arrays

Parametric/Subtyping Polymorphism

- Parametric polymorphism
 - > Parameters that can be of any valid types
- Subtyping polymorphism
 - > Parameters of certain types that can be substituted with any subtypes of them
- Having both polymorphisms provides:
 - > Greater precision of type checking
 - > Reduced code duplication
- Naïve integration makes an unsound type system

Types Example: Existing Practice

```
trait List[A]
  cons(x: A): List[A] = ConsCell(x, self)
end

object ConsCell[A](first: A, rest: List[A])
  extends { List[A] }
end

object Empty extends { List[A] }      (* problematic *)

end

foo: List[Number] = Empty.cons(3).cons(√2)
```

Types Example: Fortress Type System

```
trait List[A] extends List[B] where { A extends B }  
  cons(x: B): List[B] = ConsCell(x, self)  
end
```

```
object ConsCell[A](first: A, rest: List[A])  
  extends { List[A] }  
end
```

```
object Empty extends { List[A] }  
  where { A extends Object }  
end
```

```
foo: List[Number] = Empty.cons(3).cons(√2)
```

- Type system can regeneralize at each step, statically
- Don't need to copy the list before generalizing
- Expressiveness of dynamic type system with performance benefits of static type system

Types with “where” Clauses

- “where” clauses allows:
 - > Hidden type parameters
 - > Types that have infinitely many supertypes
 - > Variant subtyping including covariant collections with read/write accesses
- Our static type system can encode data types usually considered the province of dynamic type systems
- We have completed a soundness proof for the associated type calculus

Outline

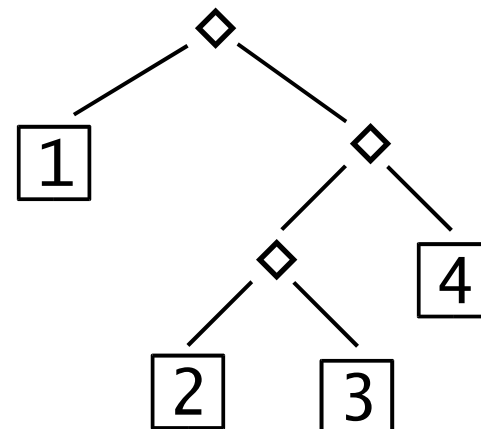
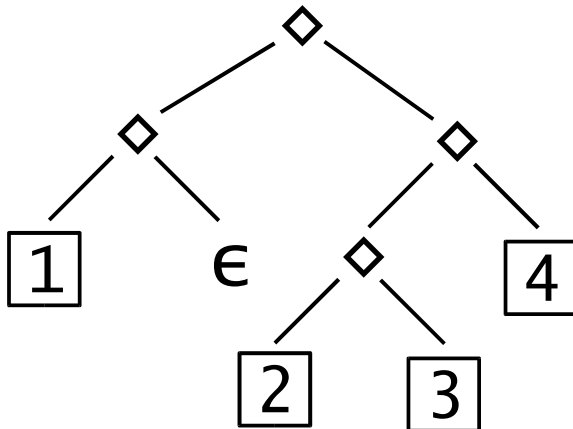
- The Fortress Programming Language
- Formalism in Fortress
 - > Formalizing Language Semantics
 - > Types Example: “where” Clauses
 - > **Types Example: Arrays**

Representation of Abstract Collections

Binary operator \diamond

Leaf operator (“unit”) \square

Optional empty collection (“zero”) ϵ
that is the identity for \diamond

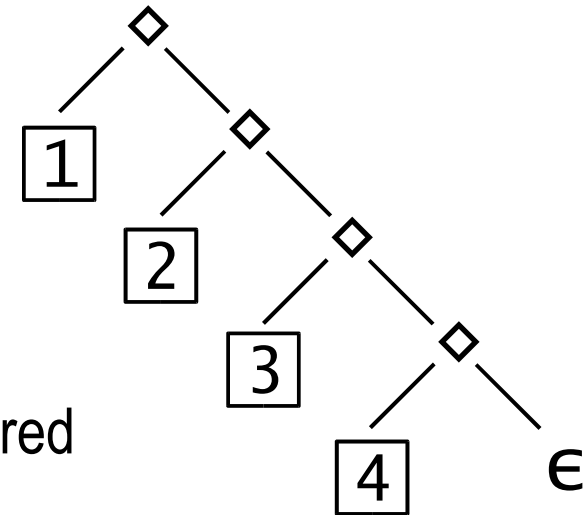
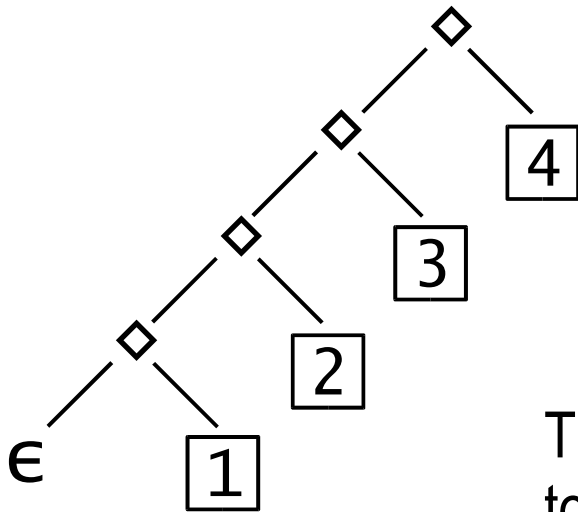
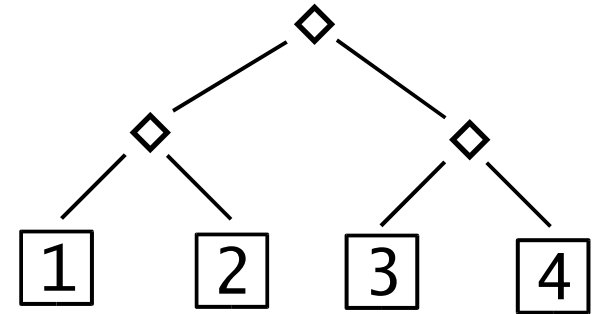
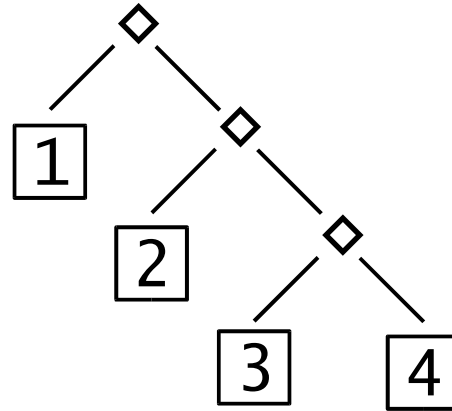
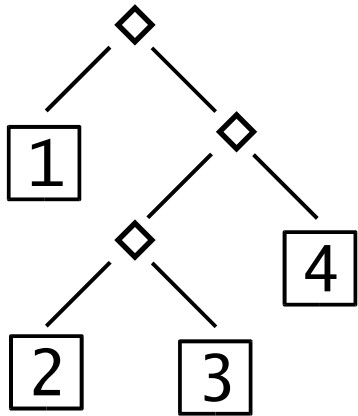


Algebraic Properties of \diamond

Associative	Commutative	Idempotent	
no	no	no	binary trees
no	no	yes	weird
no	yes	no	mobiles
no	yes	yes	weird
yes	no	no	lists
yes	no	yes	weird
yes	yes	no	multisets
yes	yes	yes	sets

The “Boom hierarchy”

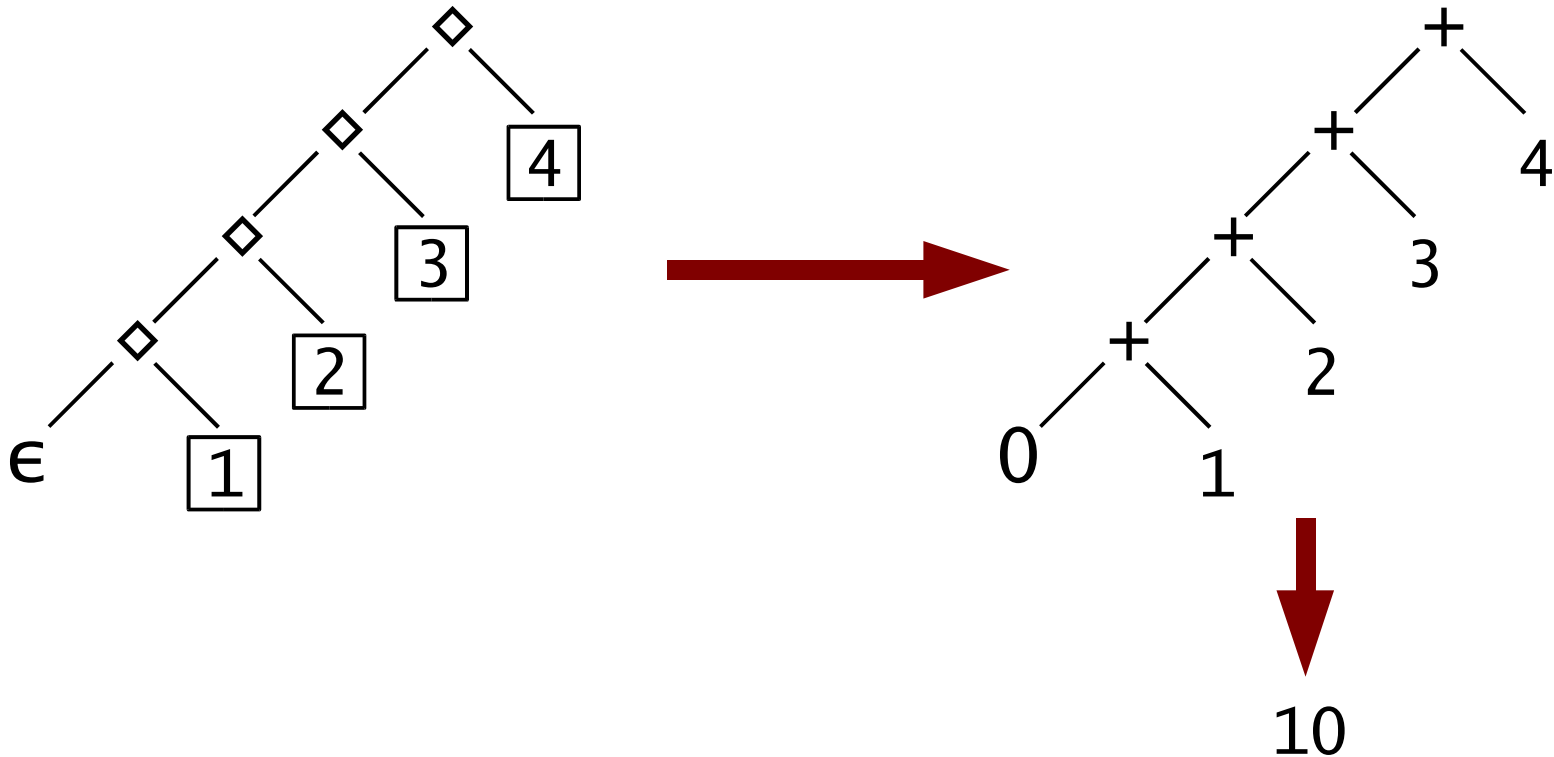
Associativity



These are all considered to be equivalent.

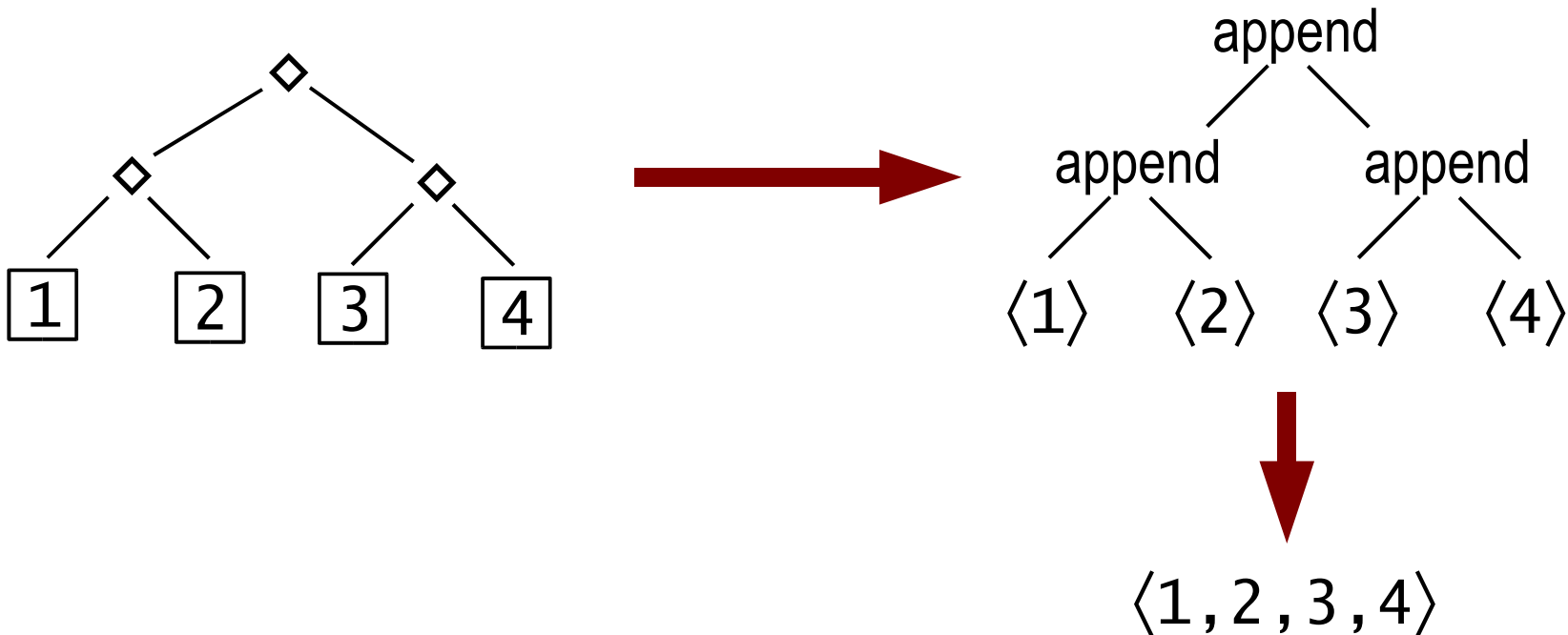
Catamorphism: Summation

Replace \diamond \square ϵ with $+$ identity 0



Catamorphism: Lists

Replace \diamond \square \in with `append` $\langle - \rangle$ \diamond



Desugaring

$\Sigma[i \leftarrow a, j \leftarrow b, p, k \leftarrow c] e$ becomes $\Sigma(f)$
 $\langle e \mid i \leftarrow a, j \leftarrow b, p, k \leftarrow c \rangle$ becomes **makeList(f)**
 where **f =**
 (fn (r)=>
 (a).generate(r, fn (i)=>
 (b).generate(r, fn (j)=>
 (p).generate(r, fn ()=>
 (c).generate(r, fn (k)=>
 r.unit(e))))))

Note: **generate** method can be overloaded!

Implementation

```
opr  $\Sigma$ [\T\](f: Catamorphism[\T,T\] $\rightarrow$ T): T
  where { T extends Monoid[\T,+,zero\] } =
  f(Catamorphism(fn(x,y) $\Rightarrow$  x+y, identity, 0))
```

```
makeList[\T\](f: Catamorphism[\T,List[\T\]\
 $\rightarrow$ List[\T\]): List[\T\] =
  f(Catamorphism(append, fn(x) $\Rightarrow$  <x>, <>))
```

Example: Lexicographic Comparison

- Assume a binary CMP operator that returns one of Less, Equal, or Greater
- Now consider the binary operator LEXICO:

LEXICO	Less	Equal	Greater
Less	Less	Less	Less
Equal	Less	Equal	Greater
Greater	Greater	Greater	Greater

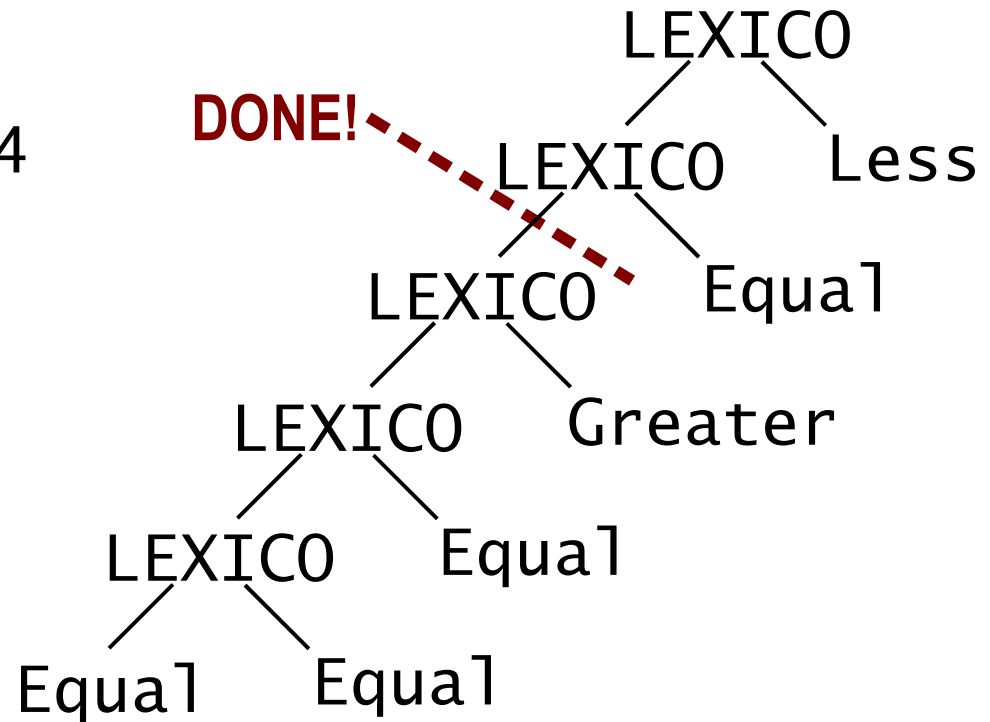
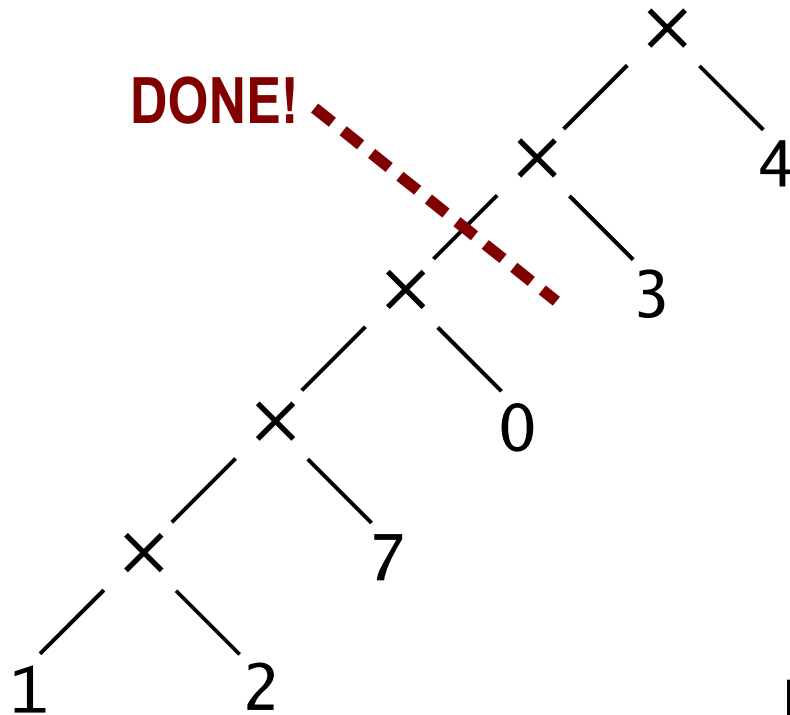
- > Associative (but *not* commutative)
- > Equal is the identity
- > Less and Greater are left zeroes

Algebraic Properties of LEXICO

```
trait Comparison extends {  
  IdentityEquality[\Comparison\  
  Associative[\Comparison, LEXICO\  
  HasRightIdentity[\Comparison, LEXICO, Equal\  
  HasLeftZeroes[\Comparison, LEXICO\  
}  
...  
test { Less, Equal, Greater }  
end
```

A generator that detects the LEXICO catamorphism (rather, the fact that it has left zeros) can choose to generate special code.

Zeros Can Stop Iteration Early



Code for Lexicographic Comparison

```

trait LexOrder[\T, E\]
  extends { TotalOrder[\T, ≤, CMP\],
            Indexable[\LexOrder[\T, E\], E\] }
  where { T extends LexOrder[\T, E\],
          E extends TotalOrder[\T, ≤, CMP\] }

  opr =(self, other:T):Boolean =
    |self| = |other| AND:
      AND[i←self.indices] self[i]=other[i]

  opr CMP(self, other:T):Comparison =
    prefix = self.indices ∩ other.indices
    (LEXICO[i←prefix] self[i] CMP other[i]) &
    LEXICO (|self| CMP |other|)

  opr ≤(self, other:T):Boolean =
    (self CMP other) ≠ Greater

end
  
```

String Comparison

```
trait String
  extends { LexOrder[\String,Character\], ... }
  ...
  test { "foo", "foobar", "quux", "" }
end
```

Parallelism in Fortress

- Aggregates used as generators drive parallelism.
- Algebraic properties drive implementation strategies.
- Algebraic properties are described by traits.
- Properties are verified by automated unit testing.
- Traits allow sharing of code, properties, and test data.
- Reducers and generators negotiate through overloaded method dispatch keyed by traits to achieve mix-and-match code selection.

Our Key Design Themes

- Make stupid mistakes impossible
And make clever mistakes relatively unlikely
- Design the language to be grown by users
Rich library language enables simple application languages
- Make abstraction efficient
Aggressive static and dynamic optimization
- Make parallelism tractable
Identify and support standard communication patterns / data types
- Emulate standard mathematical notation
Reduce the effort of translating from science to computation

sukyoung.ryu@sun.com

[http://research.sun.com/
projects/plrg](http://research.sun.com/projects/plrg)