

Parallel Programming and Parallel Abstractions in Fortress

Guy Steele

Sun Microsystems Laboratories

April 24, 2006

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

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 methods and operators
- Aggressive type inference to reduce clutter
- Aggressive static and dynamic optimization

- 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.

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.)

Data and Control Models

- Data model: shared global address space
- Control model: multithreaded
 - > Basic primitive is “spawn”
 - > We hope application code seldom uses it
- Declared distribution of data and threads
 - > Managing aggregates integrated into type system
 - > Policies programmed as libraries, not wired in
- Transactional access to shared variables
 - > Atomic blocks (implicit or explicit retry)
 - > Lock-free (no blocking, no deadlock)

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)

$$\mathbf{v_norm} = \mathbf{v} / \|\mathbf{v}\|$$

$$\Sigma[k=1:n] \mathbf{a}[k] \mathbf{x}^k$$

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

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

- Parsing this stuff is an interesting research problem

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
 - 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)
-
- Juxtaposition is a binary operator
 - Any other operator can be infix, prefix, and/or postfix
 - Many sets of brackets

Libraries Define . . .

- Which operators have infix, prefix, postfix definitions, and what types they apply to
 - opr $-(m:\text{Integer}, n:\text{Integer}) = m.\text{subtract}(n)$**
 - opr $-(m:\text{Integer}) = m.\text{negate}()$**
 - opr $(n:\text{Integer})! = \text{if } n=0 \text{ then } 1 \text{ else } n \cdot (n-1)! \text{ end}$**
- Whether a juxtaposition is meaningful
 - opr $\text{juxta}(m:\text{Integer}, n:\text{Integer}) = m.\text{times}(n)$**
- What bracketing operators actually mean
 - opr $\lceil x:\text{Number} \rceil = \text{ceiling}(x)$**
 - opr $|x:\text{Number}| = \text{if } x < 0 \text{ then } -x \text{ else } x \text{ end}$**
 - opr $|s:\text{Set}| = s.\text{size}$**

But Wasn't Operator Overloading a Disaster in C++ ?

- Yes, it was
 - > Not enough operators to go around
 - > Failure to stick to traditional meanings
- We have also been tempted and had to resist
- We see benefits in using notations for programming that are also used for specification

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 E.

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 .

$(z, norm) = \text{conjGrad}(A, x)$

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  $\llbracket$  Elt extends Number, nat N,
           Mat extends Matrix  $\llbracket$  Elt,  $N \times N$   $\rrbracket$ ,
           Vec extends Vector  $\llbracket$  Elt, N  $\rrbracket$ 
            $\rrbracket$  ( $A$ : Mat,  $x$ : Vec): (Vec, Elt)
  
```

```

cgitmax = 25
  
```

```

 $z$ : Vec := 0
  
```

```

 $r$ : Vec :=  $x$ 
  
```

```

 $p$ : Vec :=  $r$ 
  
```

```

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

```

for  $j \leftarrow$  seq(1: cgitmax) 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
  
```

```

( $z$ ,  $\|x - A z\|$ )
  
```

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 )

```

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←1:m, j←i:n 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 do a[i,j] := b[i] c[j] end
```

```
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 `<1,2,4,3,4>` 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 $\Sigma[x \leftarrow S] x$ *)

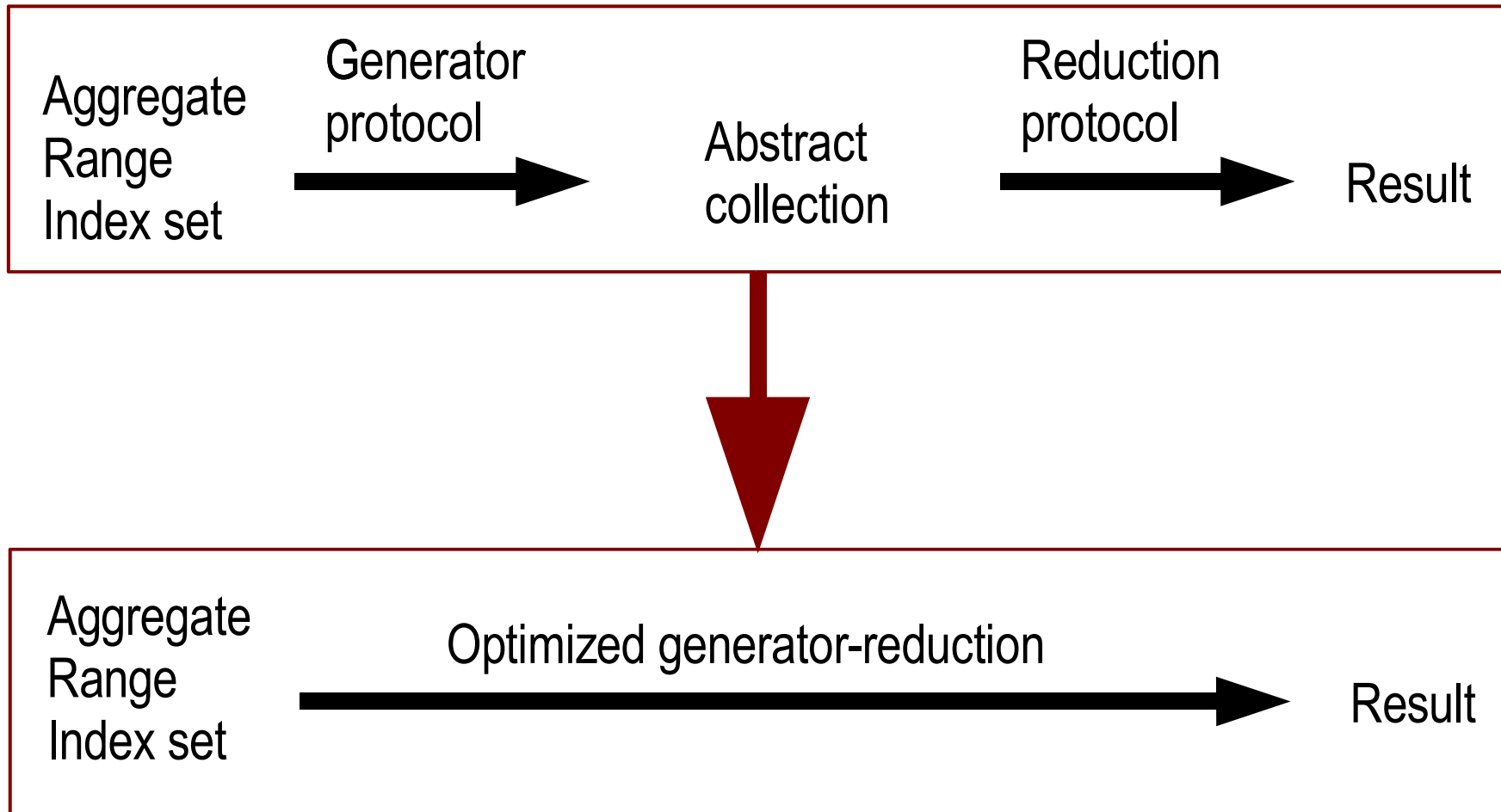
$v = \cap[k \leftarrow S, \text{prime } k] \text{arrayOfSets}[k]$

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

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

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

Abstract Collections

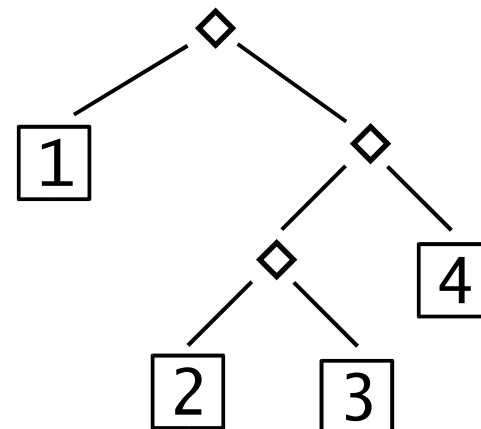
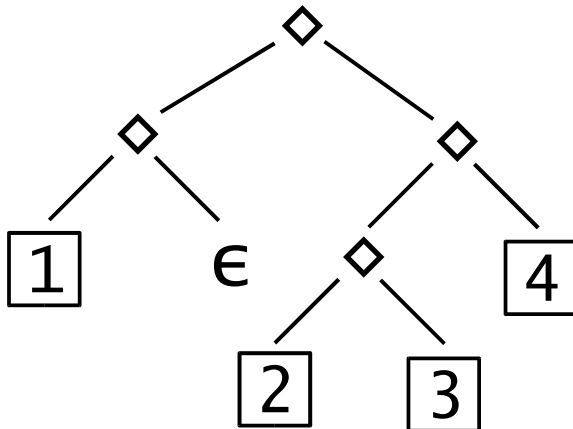


Representation of Abstract Collections

Binary operator \diamond

Leaf operator (“unit”) \square

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

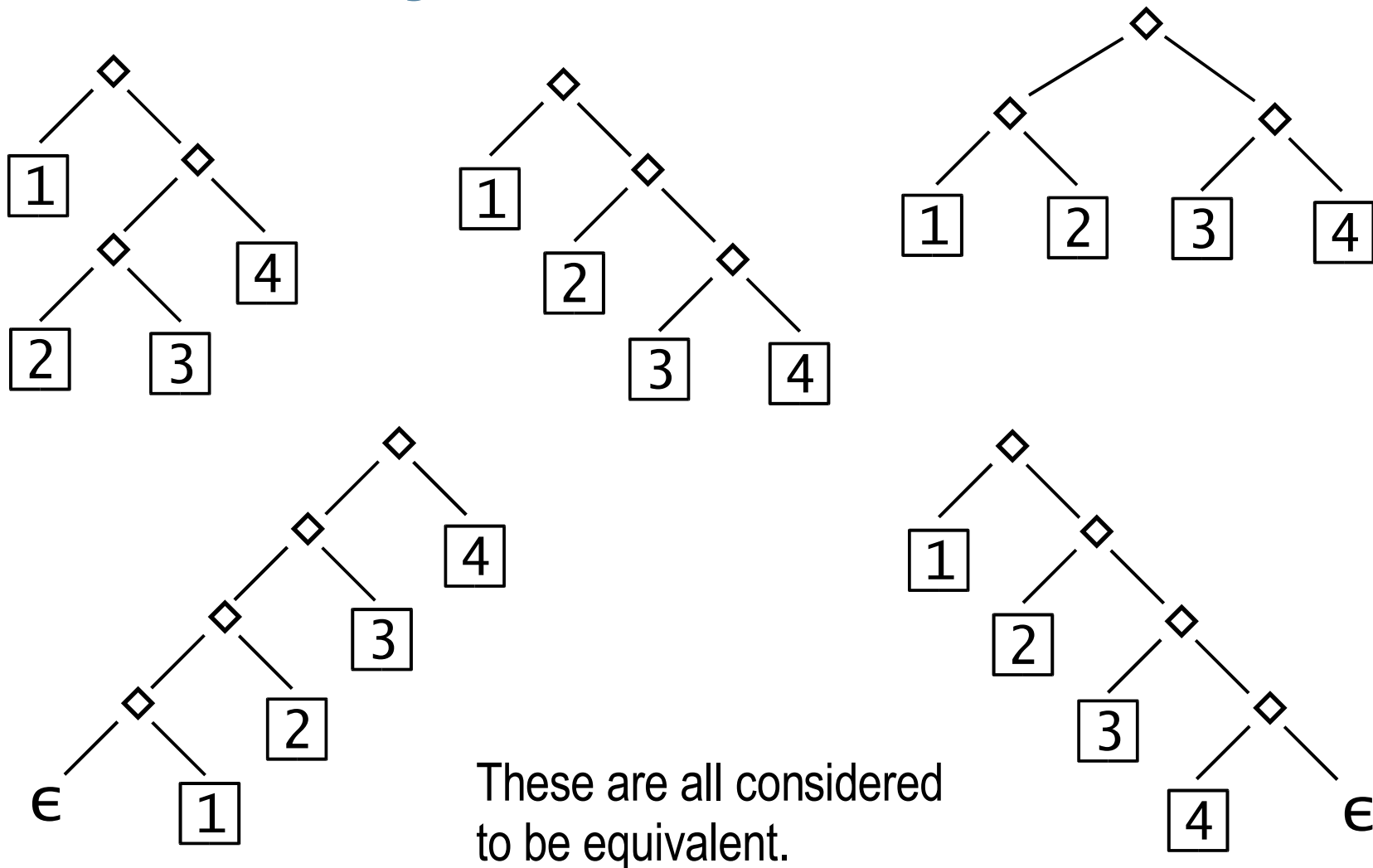


Algebraic Properties of \diamond

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

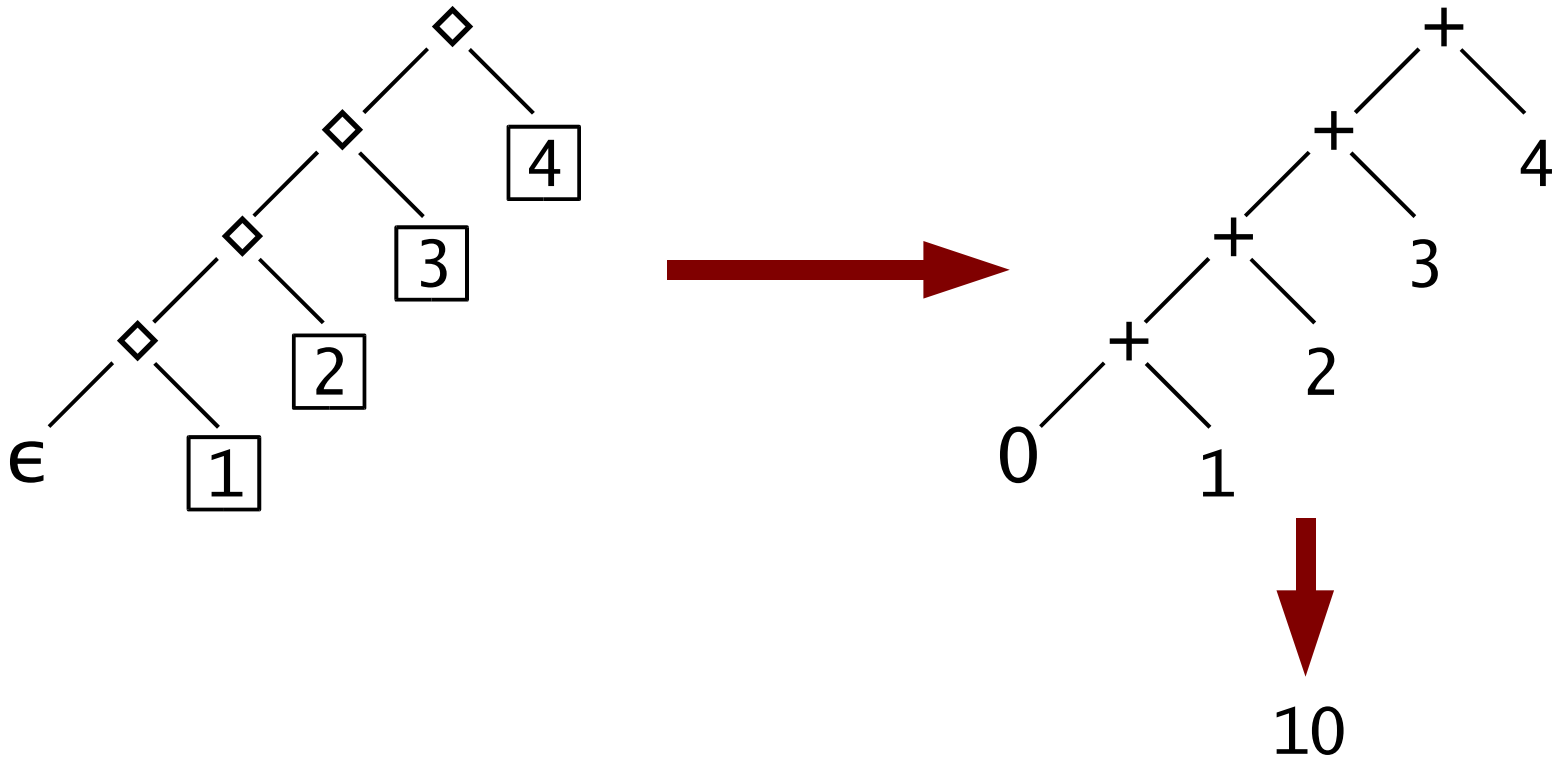
The “Boom hierarchy”

Associativity



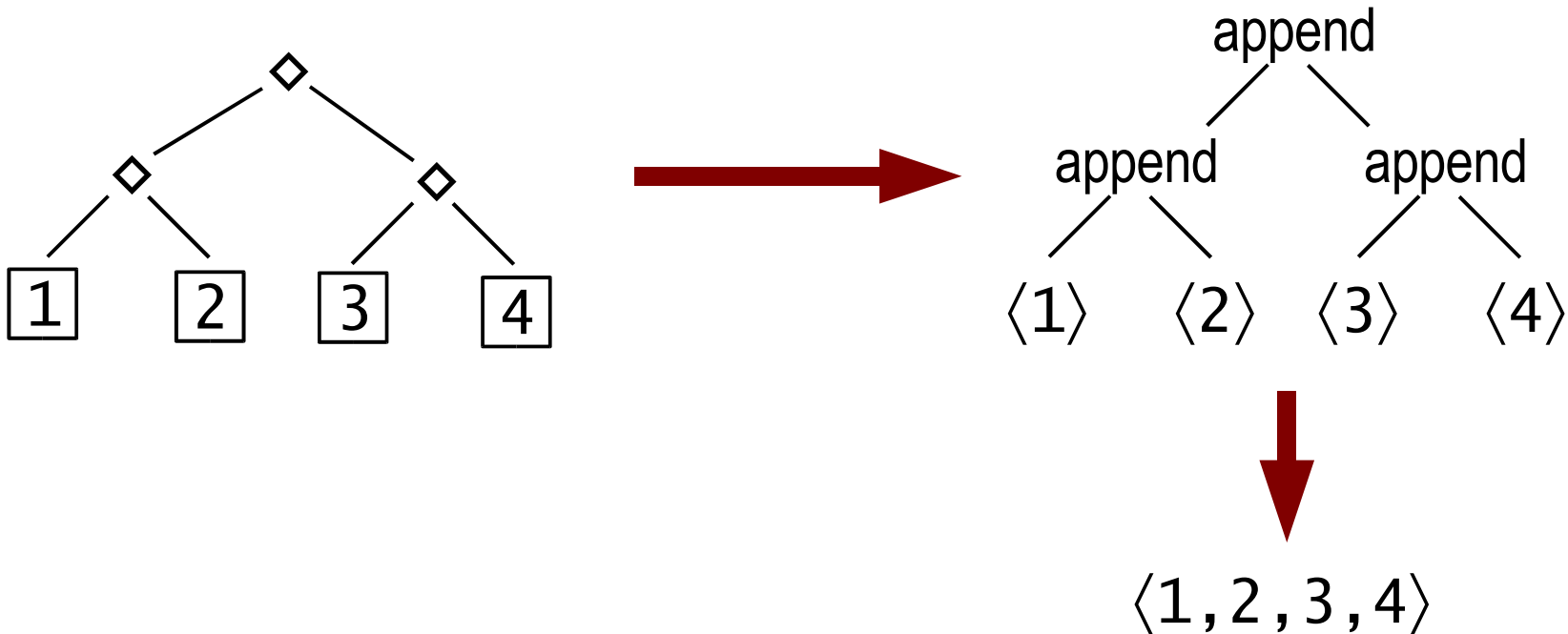
Catamorphism: Summation

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



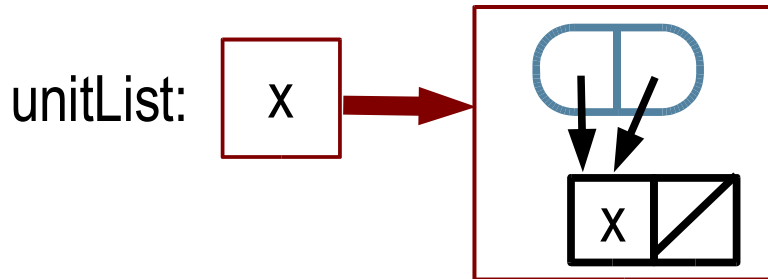
Catamorphism: Lists

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

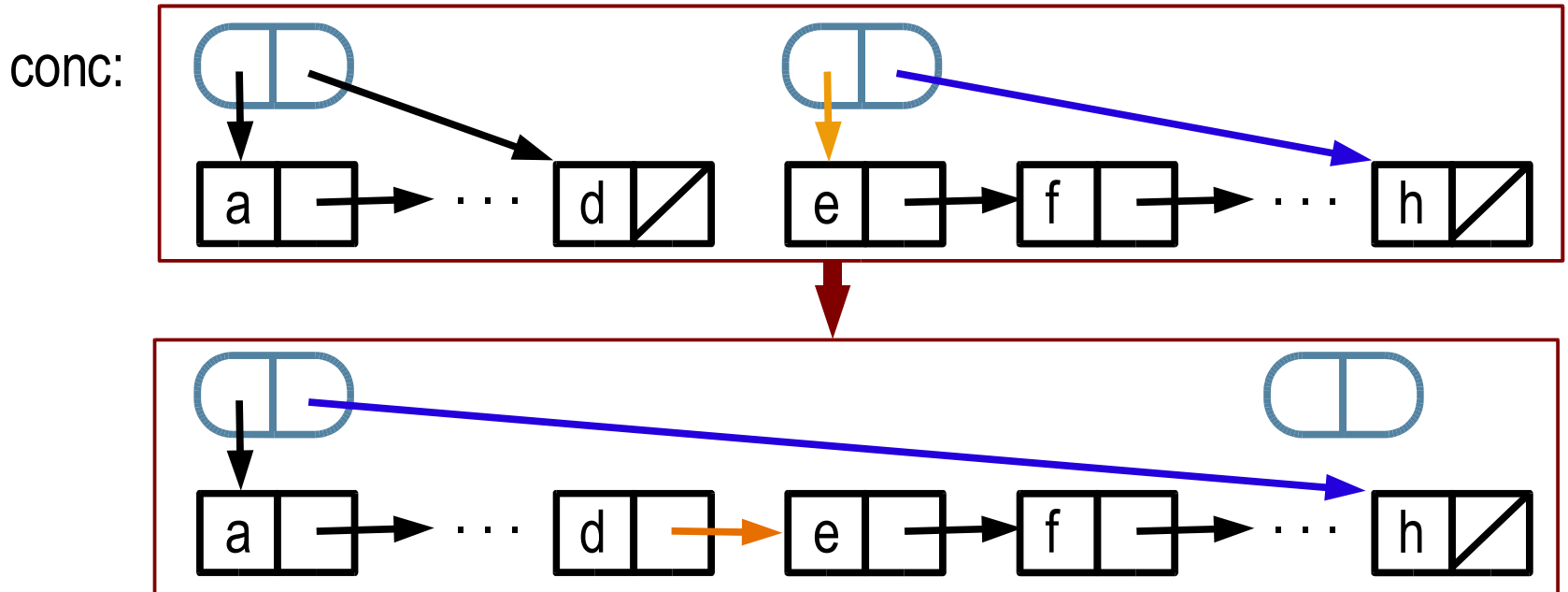


Catamorphism: Splicing Linked Lists

Replace \diamond \square ϵ with `conc` `unitList` `nil`

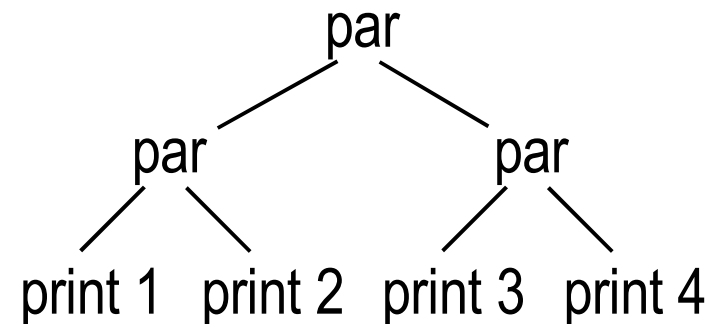
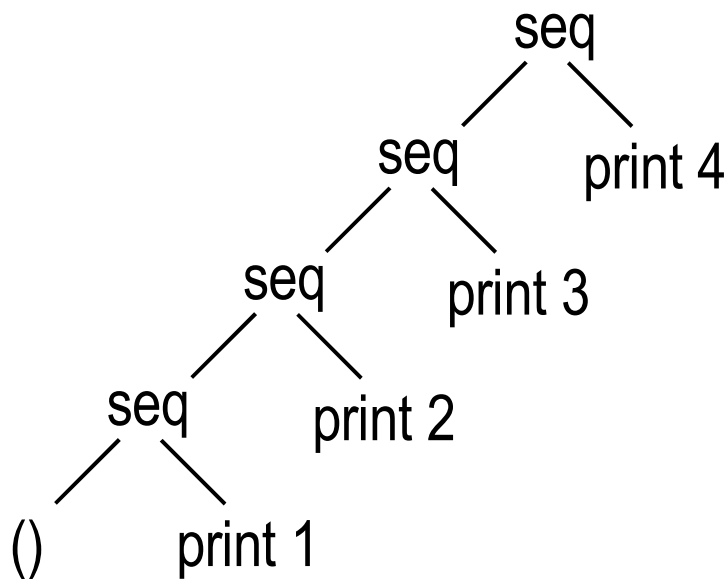


(At the end, use the left-hand pointer of the final pair.)



Catamorphism: Loops

Replace \diamond \square \in with seq identity $()$ or par identity $()$
 where $\text{seq}: (), () \rightarrow ()$ and $\text{par}: (), () \rightarrow ()$



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	List(f)
for $i \leftarrow a, j \leftarrow b, p, k \leftarrow c$ do e end	becomes	For(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))
```

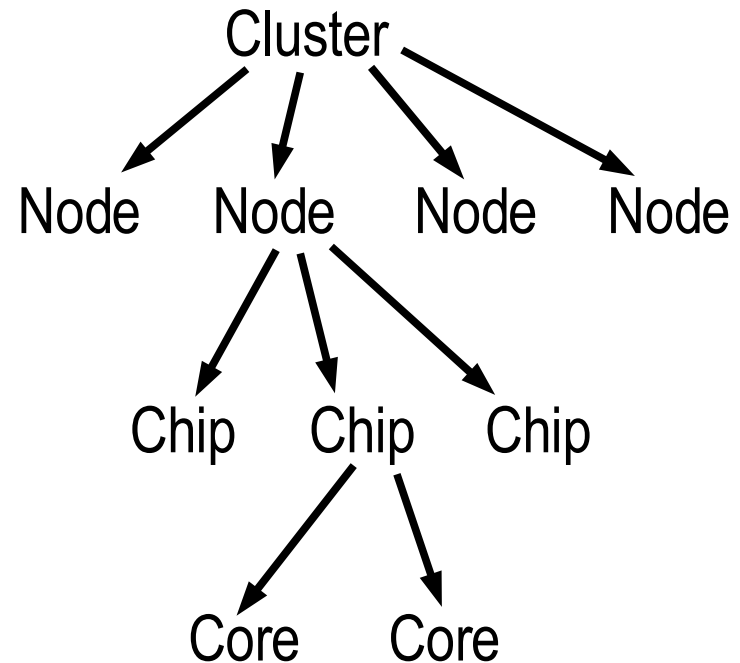
```
List[\T\](f: Catamorphism[\T,List[\T\]\
 $\rightarrow$ List[\T\]): List[\T\] =
  f(Catamorphism(append, fn(x) $\Rightarrow$  <x>, <>))
```

```
List[\T\](f: Catamorphism[\T,List[\T\]\
 $\rightarrow$ List[\T\]): List[\T\] =
  f(Catamorphism(conc, unitList, nil)).first
```

```
For(f: Catamorphism[\(),()\math>\rightarrow()\): () =
  f(Catamorphism(par, identity, ()))
```

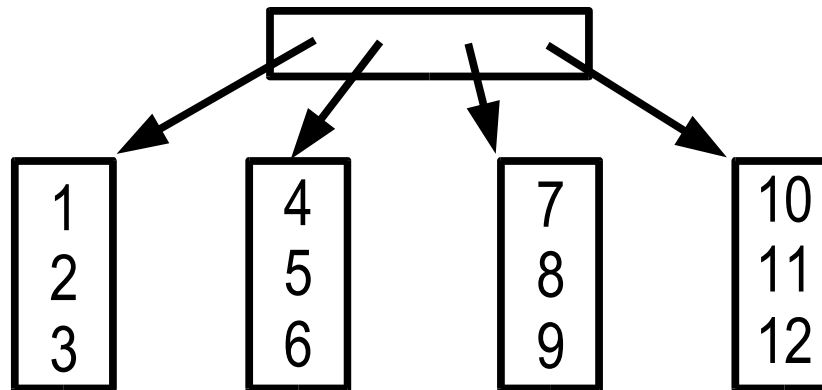
Regions

- Hierarchical data structure describes CPU and memory resources and their properties
 - > Allocation heaps
 - > Parallelism
 - > Memory coherence
- A running thread can find out its resources
- Spawn takes an optional region argument

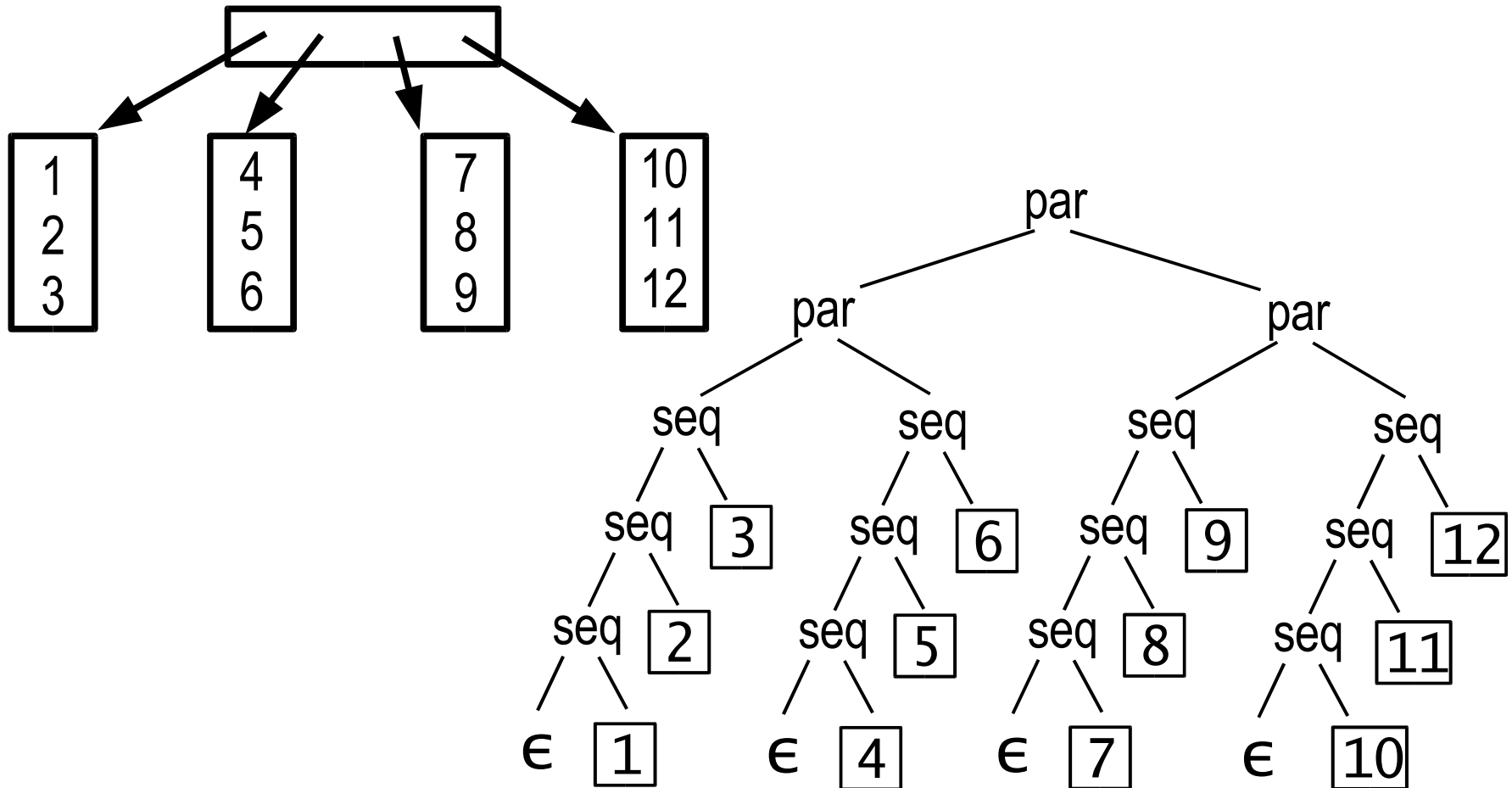


Distributions

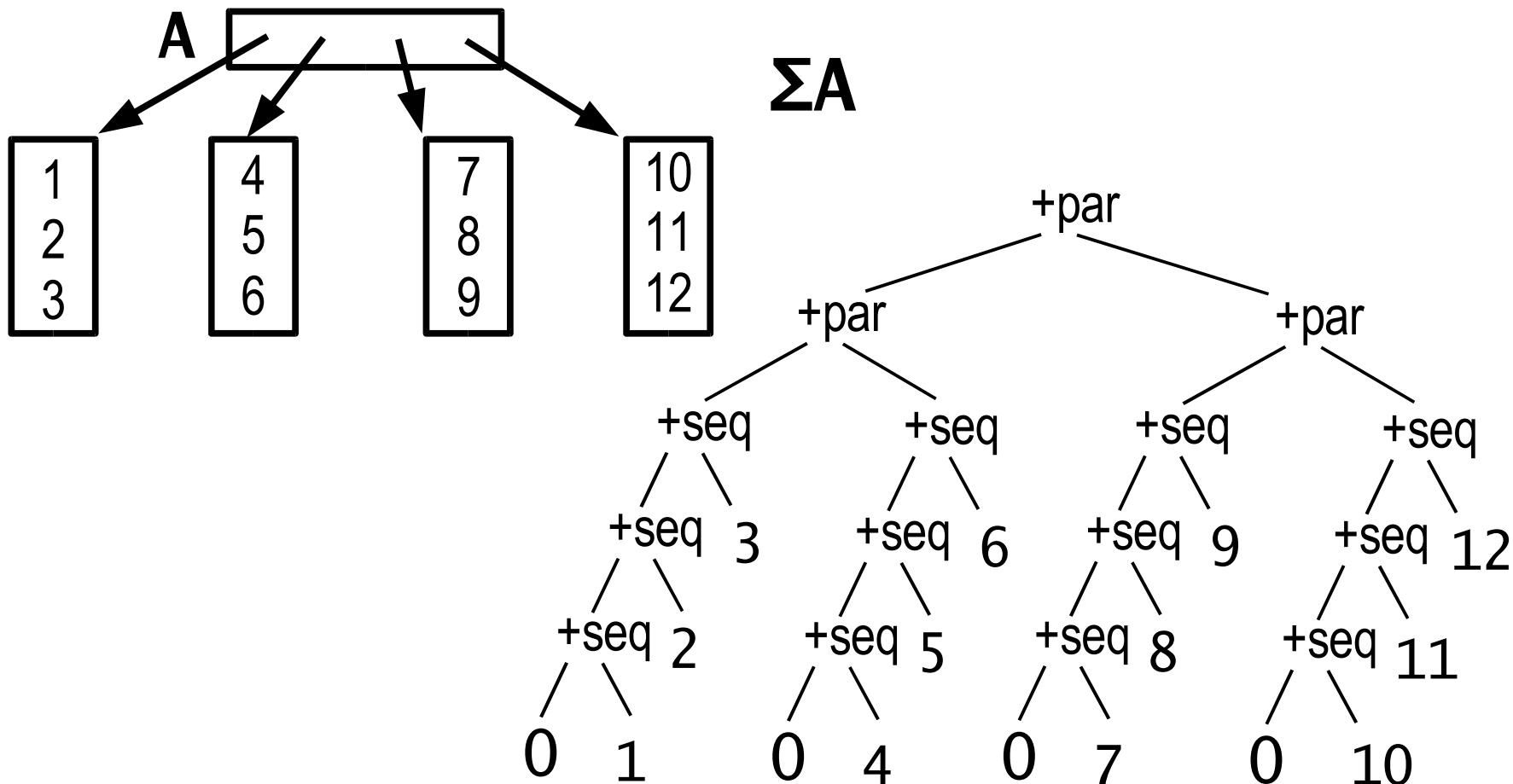
- Describe how to map a data structure onto a region
 - > Block, block-cyclic, etc., and user-definable!
 - > Map an array into a chip? Use a local heap.
 - > Map an array onto a cluster? Break it up.



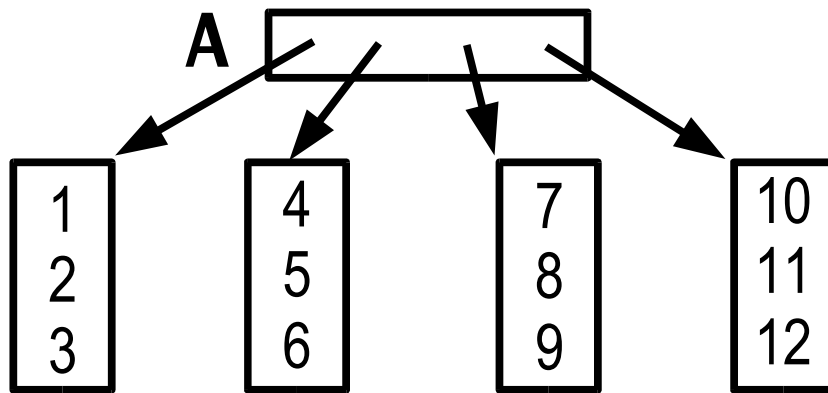
Generators Drive Parallelism



Generators Modify Reducers: Parallelism



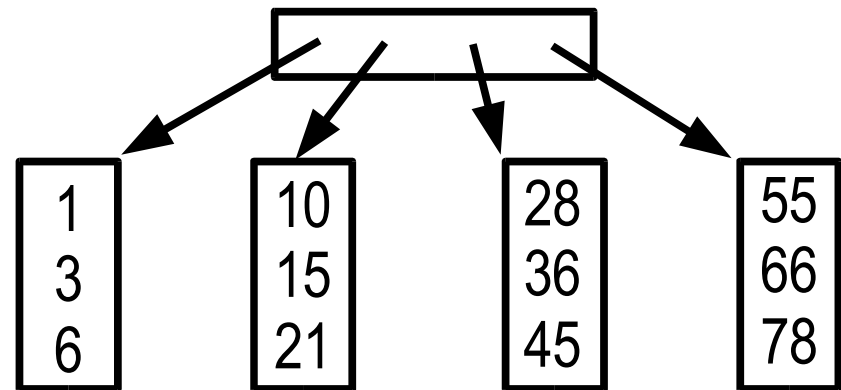
Generators Modify Reducers: Distribution



$$[\ x(x+1)/2 \ | \ x \leftarrow A \]$$

Generators and reducers are not required to use the binary-operator catamorphism protocol.

They may agree to use another protocol that, for example, communicates array shapes and distribution information.



More Desugaring

$[e \mid i \leftarrow a, j \leftarrow b, p, k \leftarrow c] [d]$ becomes **Array(f, d)**

$[e \mid i \leftarrow a, j \leftarrow b, p, k \leftarrow c] ^C$ becomes **C(f)**

$[e \mid i \leftarrow a, j \leftarrow b, p, k \leftarrow c] ^C [d]$ becomes **C(f, d)**

This lets us specify a distribution explicitly as a subscript, and/or a type constructor/catamorphism as a superscript.

$$\left[x(x+1)/2 \mid x \leftarrow 1:n \right]_{blockCyclic(4)}$$

$$\left[x \mid x > 0 \right]^{Maybe}$$

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

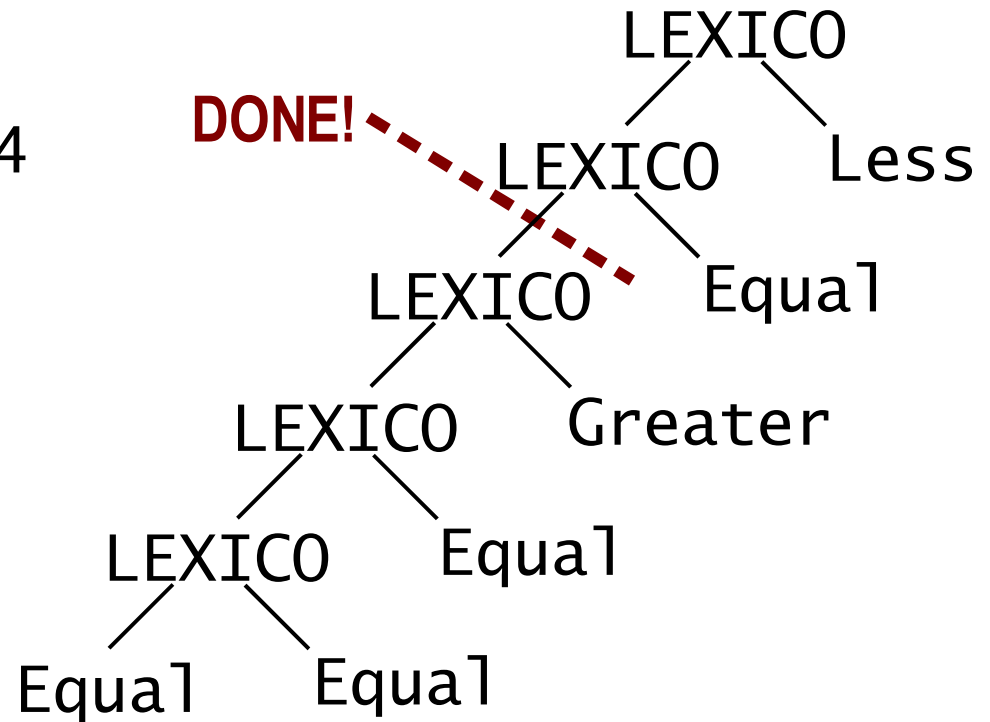
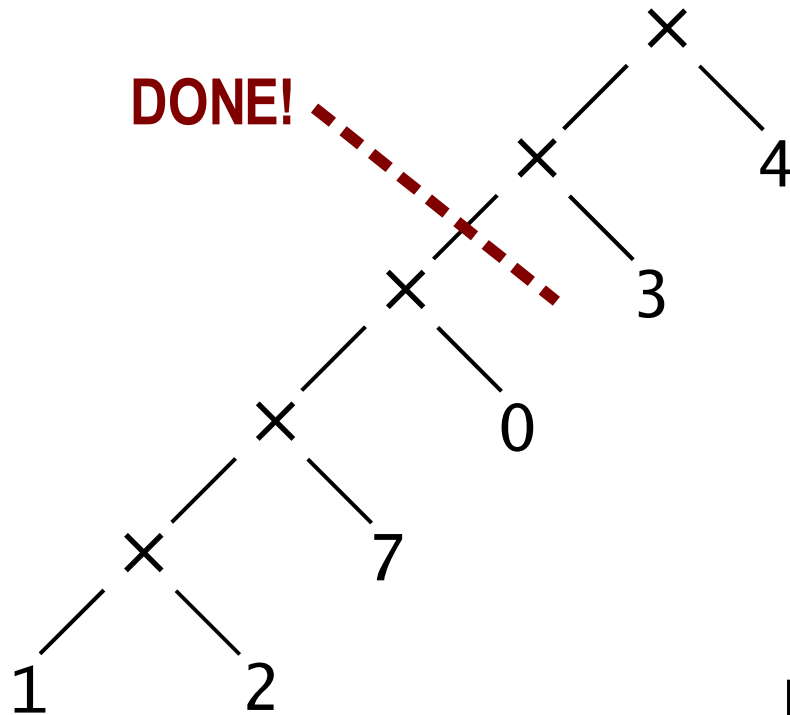
- > Associative (but *not* commutative)
- > Equal is the identity
- > Less and Greater are left zeroes

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 = do
    prefix = self.indices ∩ other.indices
    (LEXICO[i←prefix] self[i] CMP other[i]) &
    LEXICO (|self| CMP |other|)
  end

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

end

```

String Comparison

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

Summary: Parallelism in Fortress

- Regions describe machine resources.
- Distributions map aggregates onto regions.
- 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.

guy.steele@sun.com

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