Fortress
Programming Language Project Status

Guy Steele

Sun Fellow

Sun Microsystems Laboratories

September 2008

Copyright © 2008 Sun Microsystems, Inc. ("Sun"). All rights are reserved by Sun except as expressly stated as follows. Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted, provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers, or to redistribute to lists, requires prior specific written permission of Sun.

Fortress Status Report

- Fortress is a growable, mathematically oriented, parallel programming language
- Started under Sun/DARPA HPCS program, 2003–2006
- Fortress is now an open-source project with international participation
- The Fortress 1.0 release (March 2008) synchronized the specification and implementation
- Moving forward, we are growing the language and libraries and developing a compiler

With Multicore, a Profound Shift

- Parallelism is here, now, and in our faces
  - Academics have been studying it for 50 years
  - Serious commercial offerings for 25 years
  - But now it's in desktops and laptops
- Specialized expertise for science codes and databases and networking
- But soon general practitioners must go parallel

The bag of programming tricks

that has served us so well

for the last 50 years

is

the wrong way to think

going forward and

must be thrown out.

Why?

- Good sequential code minimizes total number of operations.
  - Clever tricks to reuse previously computed results.
  - Good parallel code often performs redundant operations to reduce communication.
- Good sequential algorithms minimize space usage.
  - Clever tricks to reuse storage.
  - Good parallel code often requires extra space to permit temporal decoupling.
- Sequential idioms stress linear problem decomposition.
  - Process one thing at a time and accumulate results.
  - Good parallel code usually requires multiway problem decomposition and multiway aggregation of results.

Let's Add a Bunch of Numbers

```
DO I = 1, 1000000
  SUM = SUM + X(I)
END DO
```

Can it be parallelized?

Let's Add a Bunch of Numbers

```
SUM = 0              // Oops!

DO I = 1, 1000000
  SUM = SUM + X(I)
END DO
```

Can it be parallelized?

This is already bad!

Clever compilers have to undo this.

What Does a Mathematician Say?

$$\sum_{i=1}^{1000000} x_i \quad \text{or maybe just} \quad \sum x$$

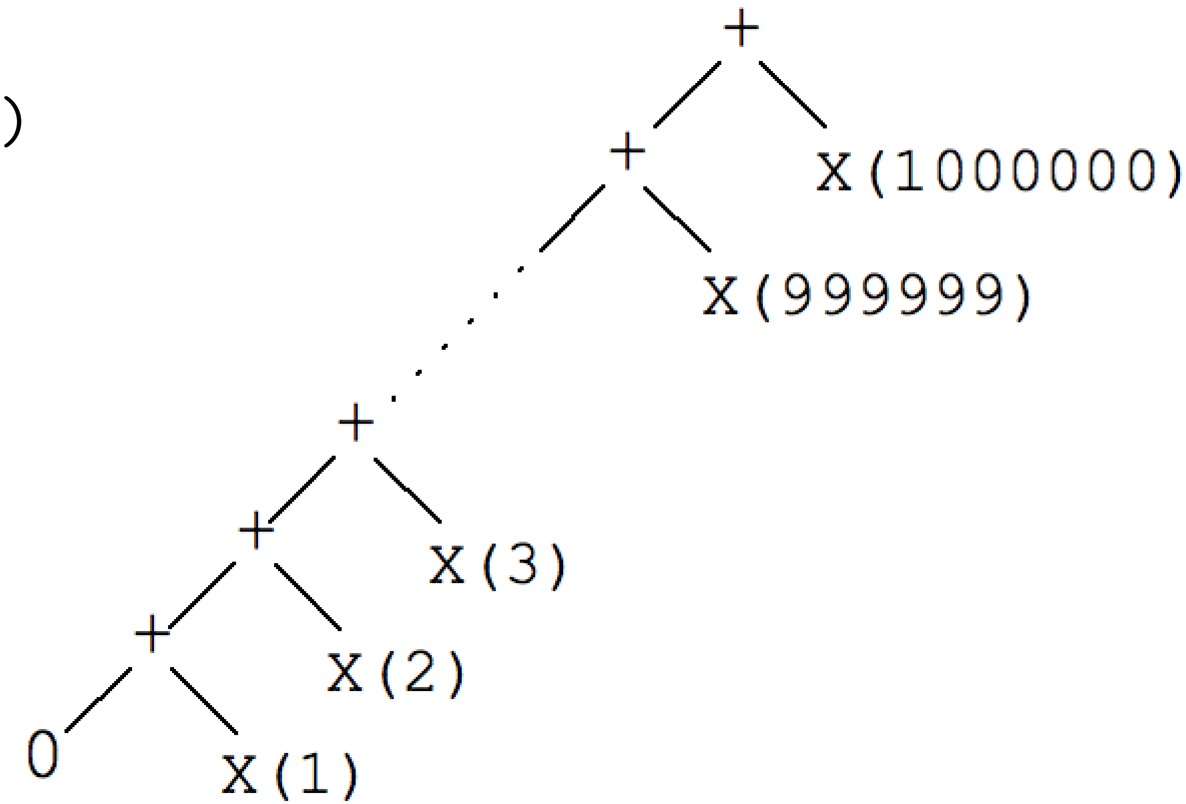
Compare Fortran 90 `SUM(X)`.

What, not how.

No commitment yet as to strategy. This is good.

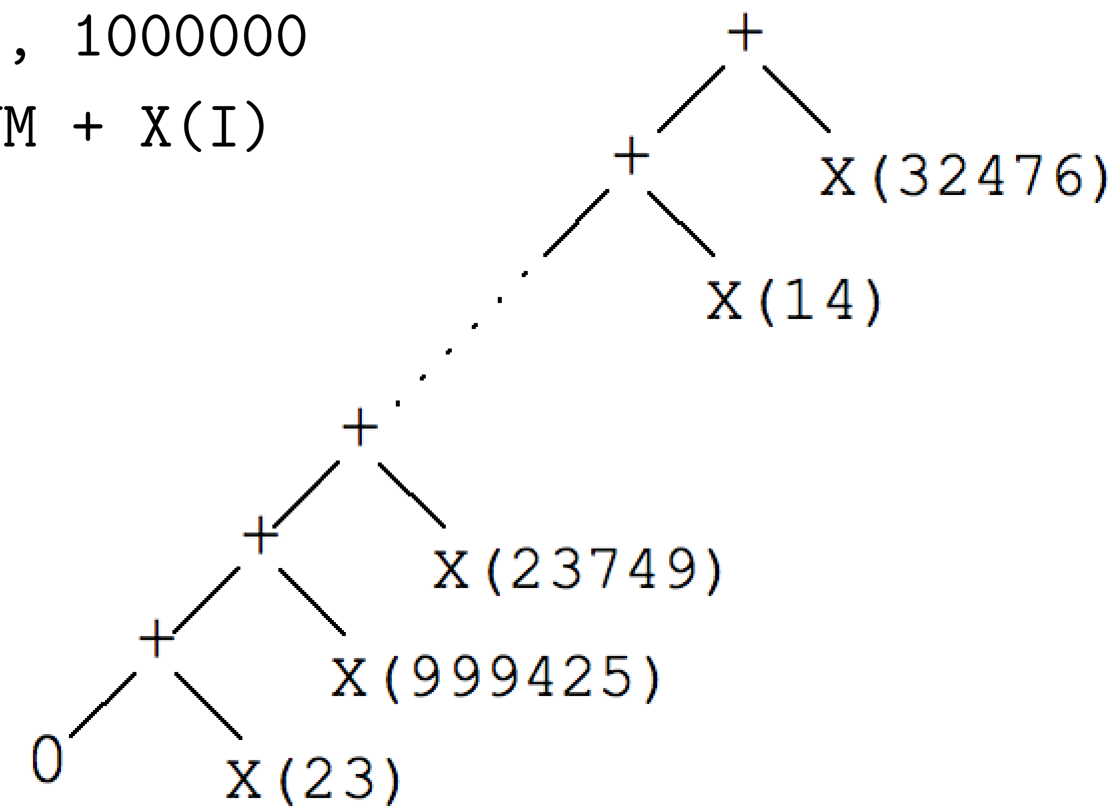
Sequential Computation Tree

```
SUM = 0
DO I = 1, 1000000
  SUM = SUM + X(I)
END DO
```



Atomic Update Computation Tree

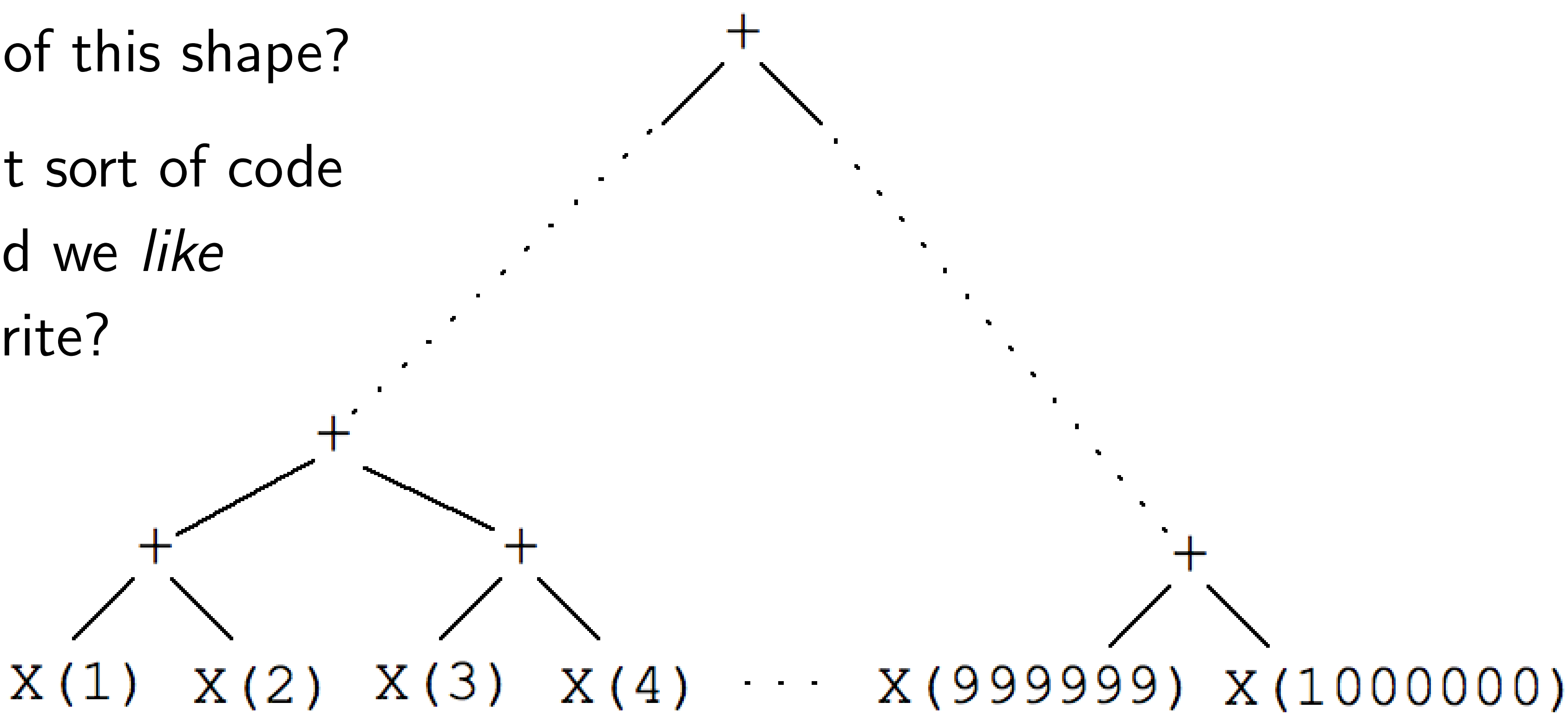
```
SUM = 0
PARALLEL DO I = 1, 1000000
  ATOMIC SUM = SUM + X(I)
END DO
```



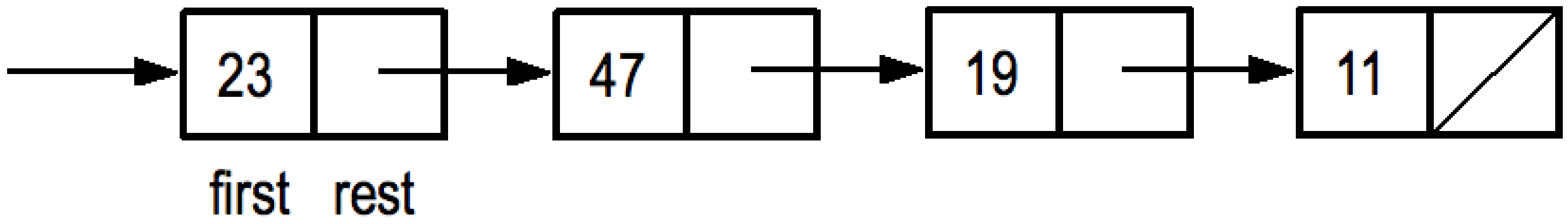
Parallel Computation Tree

What sort of code
should we write
to get a computation
tree of this shape?

What sort of code
would we *like*
to write?



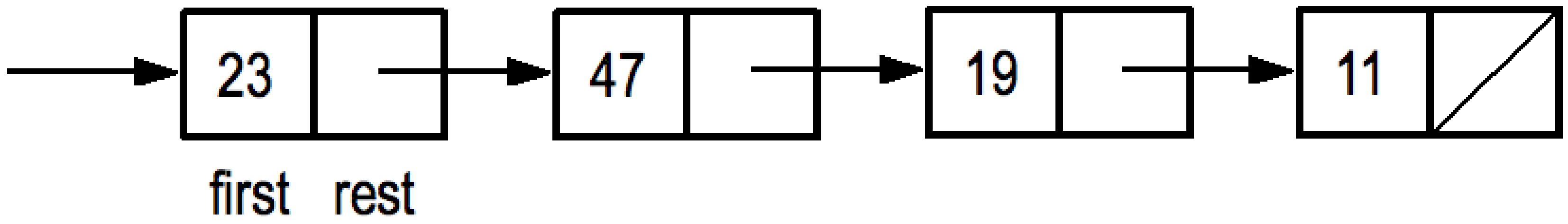
Finding the Length of a LISP List



Recursive:

```
(define length (list)
  (cond ((null list) 0)
        (else (+ 1 (length (rest list))))))
```

Finding the Length of a LISP List



Iterative:

```
(define length (list)
  (do ((x list (rest x))
       (n 0 (+ n 1)))
    ((null x) n)))
```

Length of an Object-Oriented List

```
class List<T> {
  abstract int length();
}
class Empty extends List {
  int length() { return 0; }
}
class Node<T> extends List<T> {
  T first;
  List<T> rest;
  int length() { return 1 + rest.length(); }
}
```

Linear versus Multiway Decomposition

- These are important program decomposition strategies, but inherently sequential.
  - Mostly because of the linearly organized data structure.
  - Compare Peano arithmetic: $5 = ((((0+1)+1)+1)+1)+1$
  - Binary arithmetic is much more efficient than unary!
- We need a *multiway decomposition* paradigm:

```
length [ ] = 0
length [a] = 1
length (a++b) = (length a) + (length b)
```

This is just a summation problem: adding up a bunch of 1's!

Splitting a String into Words (1)

- Given: a string
- Result: List of strings, the words separated by spaces
  - Words must be nonempty
  - Words may be separated by more than one space
  - String may or may not begin (or end) with spaces

Splitting a String into Words (2)

- Tests:

$$\mathit{println}\ \mathit{words}(\text{“}\texttt{This is a sample}\text{”})$$

$$\mathit{println}\ \mathit{words}(\text{“}\texttt{ Here is another sample }\text{”})$$

$$\mathit{println}\ \mathit{words}(\text{“}\texttt{JustOneWord}\text{”})$$

$$\mathit{println}\ \mathit{words}(\text{“}\texttt{   }\text{”})$$

$$\mathit{println}\ \mathit{words}(\text{“”})$$

- Expected output:

$$\langle\, \texttt{This}, \texttt{is}, \texttt{a}, \texttt{sample} \,\rangle$$

$$\langle\, \texttt{Here}, \texttt{is}, \texttt{another}, \texttt{sample} \,\rangle$$

$$\langle\, \texttt{JustOneWord} \,\rangle$$

$$\langle\ \rangle$$

$$\langle\ \rangle$$

Splitting a String into Words (3)

$words(s\colon \mathrm{String}) = $ `do`
$\quad result\colon \mathrm{List}[\![\mathrm{String}]\!] := \langle\,\rangle$
$\quad word\colon \mathrm{String} := \text{“”}$
$\quad$ `for` $k \leftarrow seq(0\,\#\,length(s))$ `do`
$\quad\quad char = substring(s, k, k+1)$
$\quad\quad$ `if` $(char = \text{“ ”})$ `then`
$\quad\quad\quad$ `if` $(word \neq \text{“”})$ `then` $result := result \,\|\, \langle\, word \,\rangle$ `end`
$\quad\quad\quad word := \text{“”}$
$\quad\quad$ `else`
$\quad\quad\quad word := word \,\|\, char$
$\quad\quad$ `end`
$\quad$ `end`
$\quad$ `if` $(word \neq \text{“”})$ `then` $result := result \,\|\, \langle\, word \,\rangle$ `end`
$\quad result$
`end`

Splitting a String into Words (4)

```
Here is a sesquipedalian string of words

Here is a sesquipedalian string of words

Here is a sesquipedalian string of words
```

Splitting a String into Words (5)

$$maybeWord(s\colon \mathrm{String})\colon \mathrm{List}[\![\mathrm{String}]\!] =$$

$$\quad \texttt{if } s = \text{``''} \texttt{ then } \langle\,\rangle \texttt{ else } \langle\, s \,\rangle \texttt{ end}$$

$$\texttt{trait } \mathrm{WordState}$$
$$\qquad \texttt{extends } \{\, \mathrm{Associative}[\![\mathrm{WordState}, \oplus]\!] \,\}$$
$$\qquad \texttt{comprises } \{\, \mathrm{Chunk}, \mathrm{Segment} \,\}$$
$$\quad \texttt{opr } \oplus(\texttt{self}, \mathit{other}\colon \mathrm{WordState})\colon \mathrm{WordState}$$
$$\texttt{end}$$

Splitting a String into Words (6)

$$
\begin{array}{l}
\texttt{object}\ \ \mathrm{Chunk}(s\colon \mathrm{String})\ \texttt{extends}\ \ \mathrm{WordState} \\
\quad \texttt{opr}\ \oplus(\texttt{self}, \mathit{other}\colon \mathrm{Chunk})\colon \mathrm{WordState} = \mathrm{Chunk}(s \parallel \mathit{other}.s) \\
\quad \texttt{opr}\ \oplus(\texttt{self}, \mathit{other}\colon \mathrm{Segment})\colon \mathrm{WordState} = \\
\qquad \mathrm{Segment}(s \parallel \mathit{other}.l, \mathit{other}.A, \mathit{other}.r) \\
\texttt{end}
\end{array}
$$

$$
\begin{array}{l}
\texttt{object}\ \ \mathrm{Segment}(l\colon \mathrm{String}, A\colon \mathrm{List}[\![\mathrm{String}]\!], r\colon \mathrm{String}) \\
\qquad \texttt{extends}\ \ \mathrm{WordState} \\
\quad \texttt{opr}\ \oplus(\texttt{self}, \mathit{other}\colon \mathrm{Chunk})\colon \mathrm{WordState} = \\
\qquad \mathrm{Segment}(l, A, r \parallel \mathit{other}.s) \\
\quad \texttt{opr}\ \oplus(\texttt{self}, \mathit{other}\colon \mathrm{Segment})\colon \mathrm{WordState} = \\
\qquad \mathrm{Segment}(l, A \parallel \mathit{maybeWord}(r \parallel \mathit{other}.l) \parallel \mathit{other}.A, \mathit{other}.r) \\
\texttt{end}
\end{array}
$$

Splitting a String into Words (7)

$$
\begin{array}{l}
\mathit{processChar}(c\colon \mathrm{String})\colon \mathrm{WordState} = \\
\quad \texttt{if } (c = \text{`` ''}) \texttt{ then } \mathrm{Segment}(\text{``''}, \langle\,\rangle, \text{``''}) \\
\quad \texttt{else } \mathrm{Chunk}(c) \\
\quad \texttt{end}
\end{array}
$$

$$
\begin{array}{l}
\mathit{words}(s\colon \mathrm{String}) = \texttt{do} \\
\quad g = \bigoplus_{k \leftarrow 0 \# \mathit{length}(s)} \mathit{processChar}(\mathit{substring}(s, k, k+1)) \\
\quad \texttt{typecase } g \texttt{ of} \\
\qquad \mathrm{Chunk} \Rightarrow \mathit{maybeWord}(g.s) \\
\qquad \mathrm{Segment} \Rightarrow \mathit{maybeWord}(g.l) \,\|\, g.A \,\|\, \mathit{maybeWord}(g.r) \\
\quad \texttt{end} \\
\texttt{end}
\end{array}
$$

What's Going On Here?

Instead of linear induction
with one base case (empty),
we have multiway induction
with two base cases (empty and unit).

Why are these two base cases important?

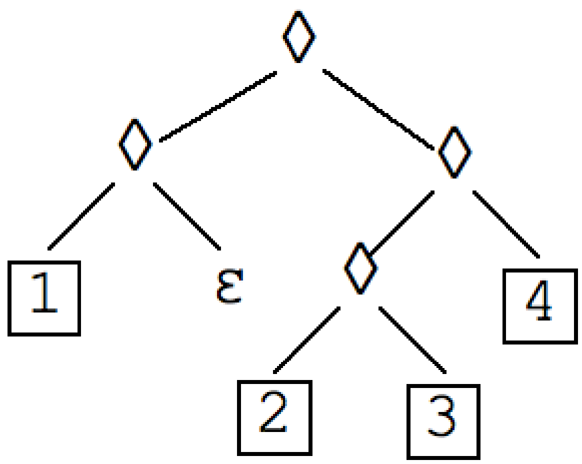
Representation of Abstract Collections

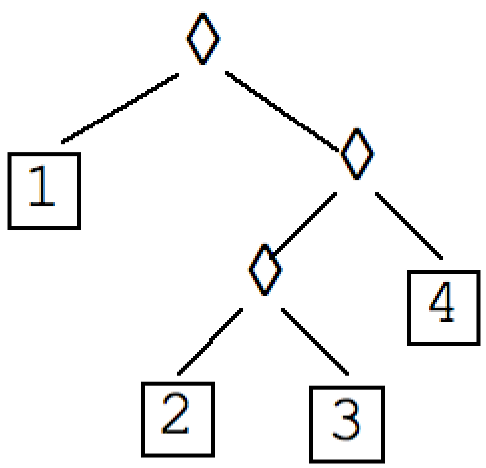
Binary operator ◊

Leaf operator ("unit") □

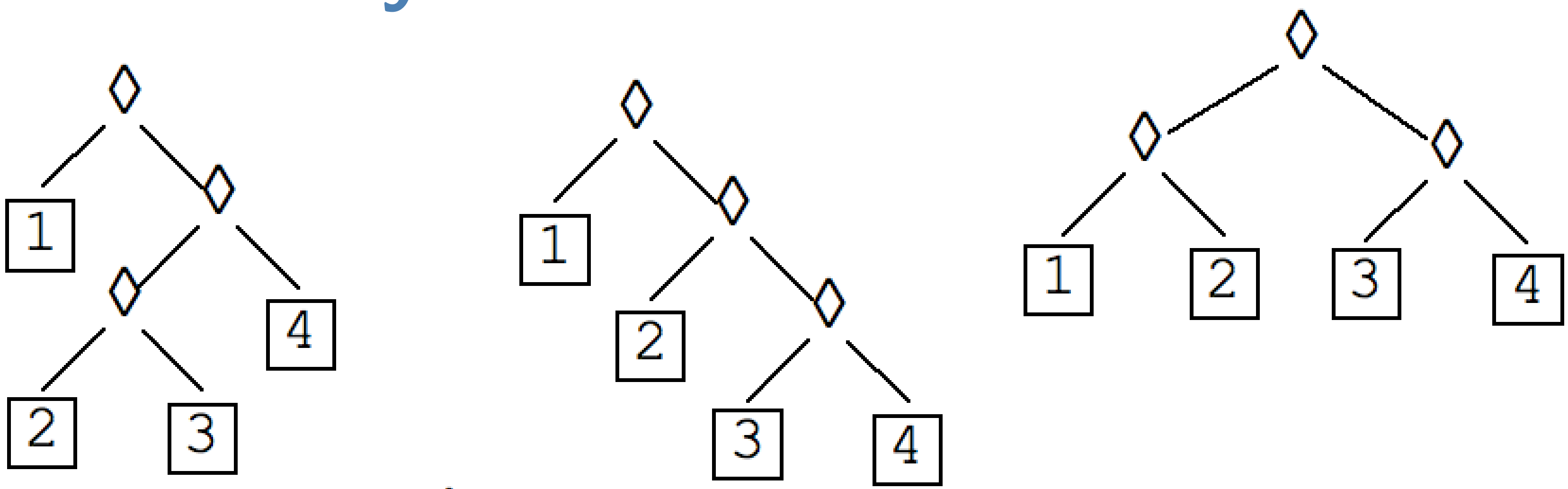
Optional empty collection ("zero") ε

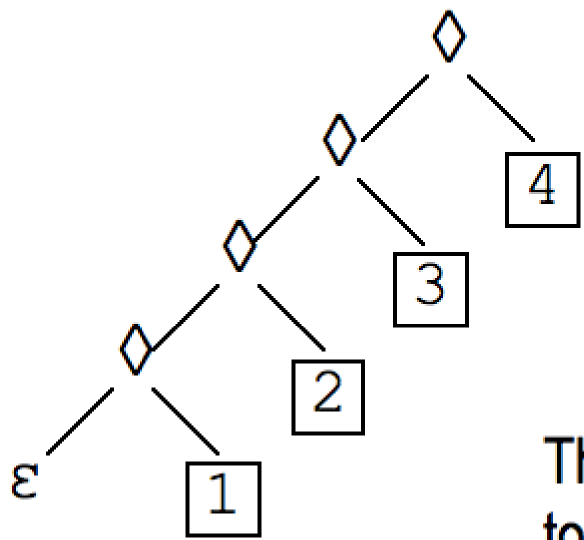
that is the identity for ◊



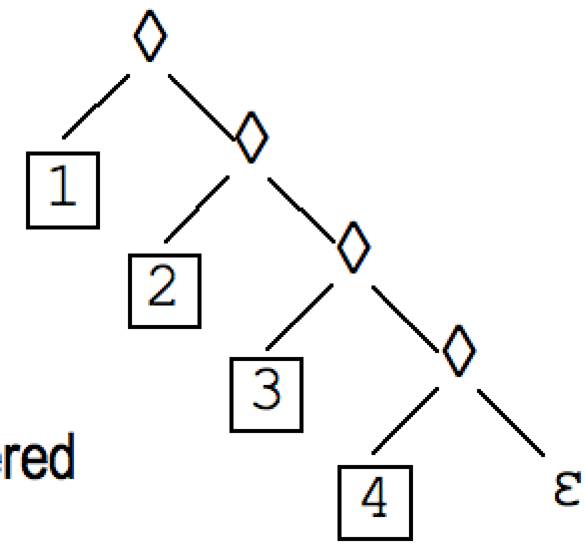


Associativity



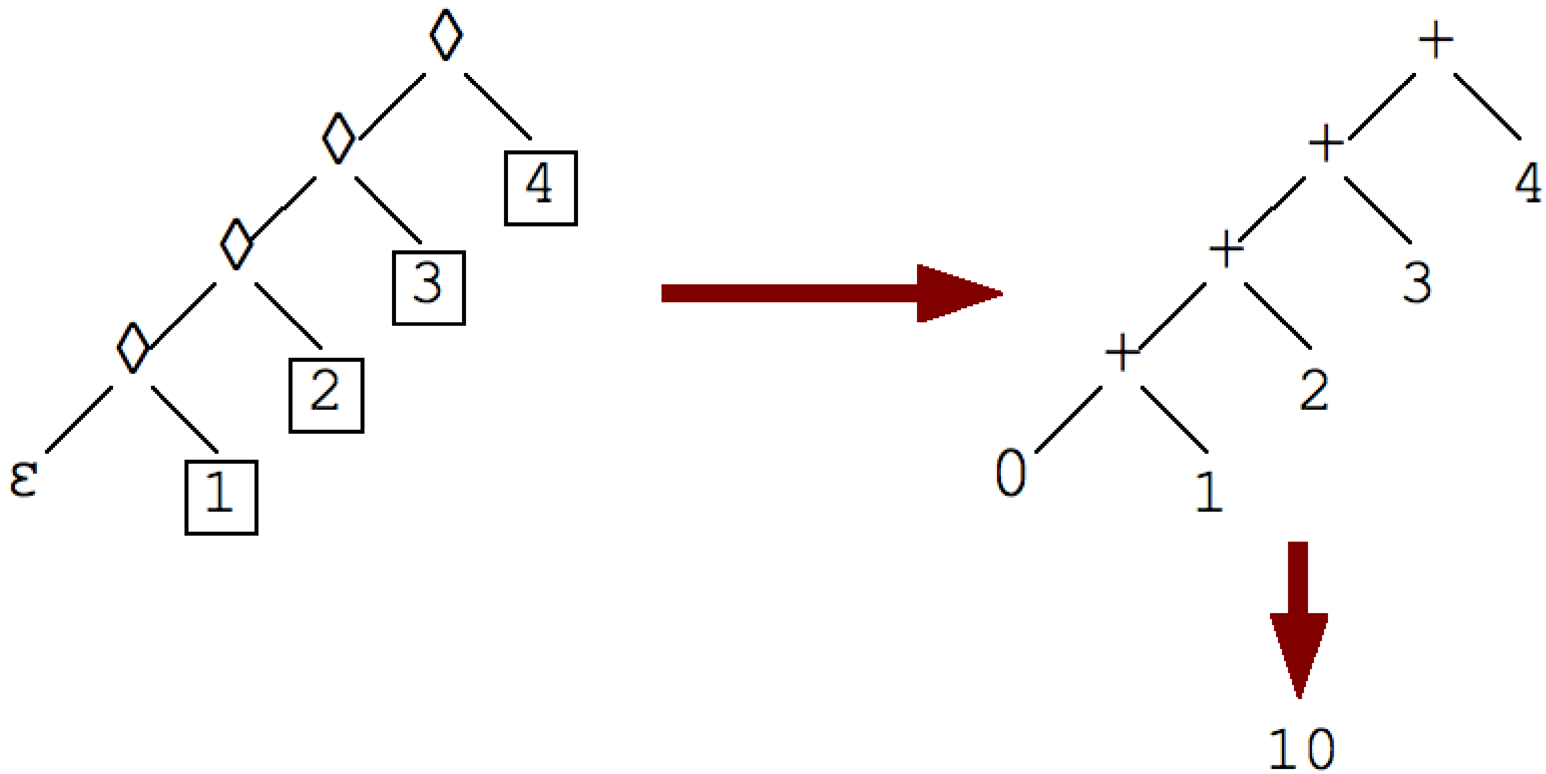


These are all considered to be equivalent.



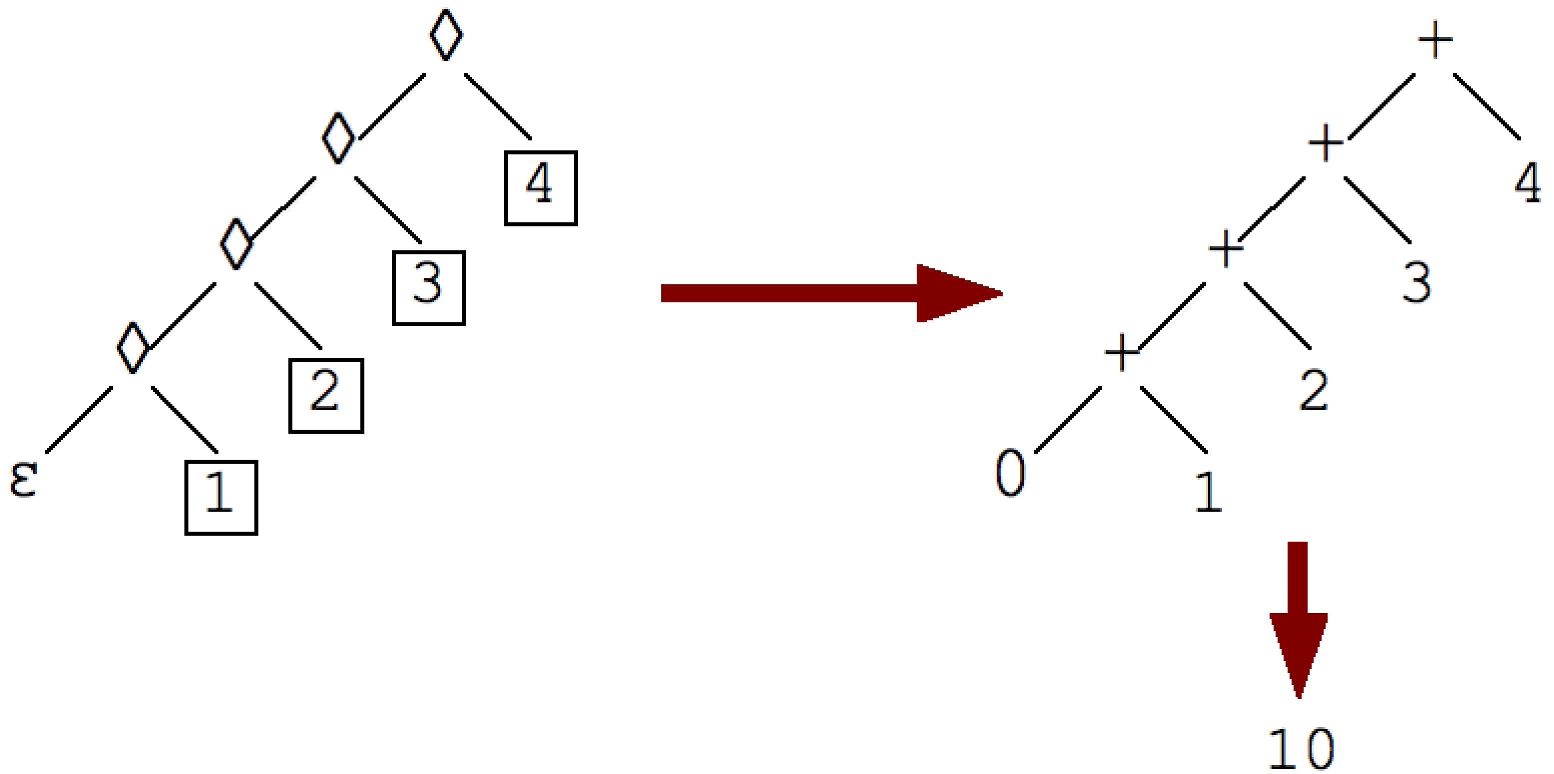
Catamorphism: Summation

Replace ◊ □ ε with + identity 0



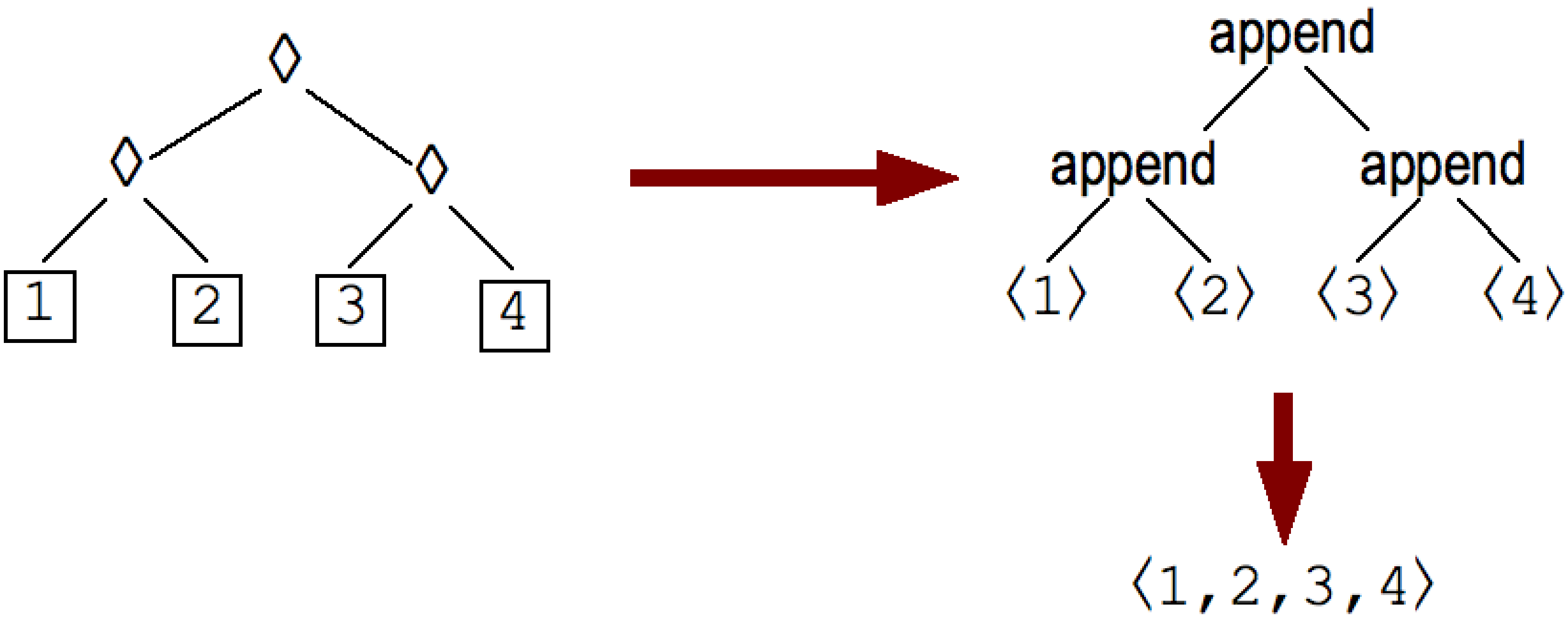
Computation: Summation

Replace ◊ □ ε with + identity 0



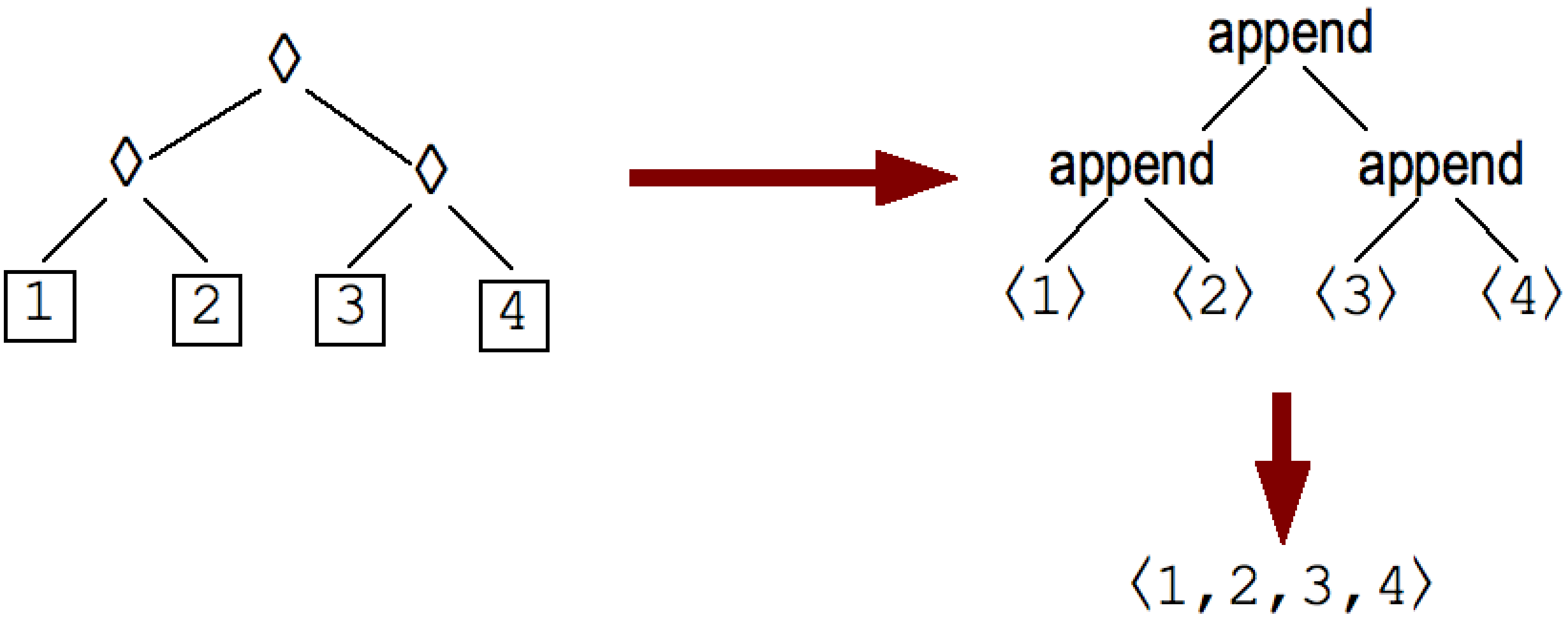
Catamorphism: Lists

Replace ◊ □ ε with append ⟨–⟩ ⟨⟩



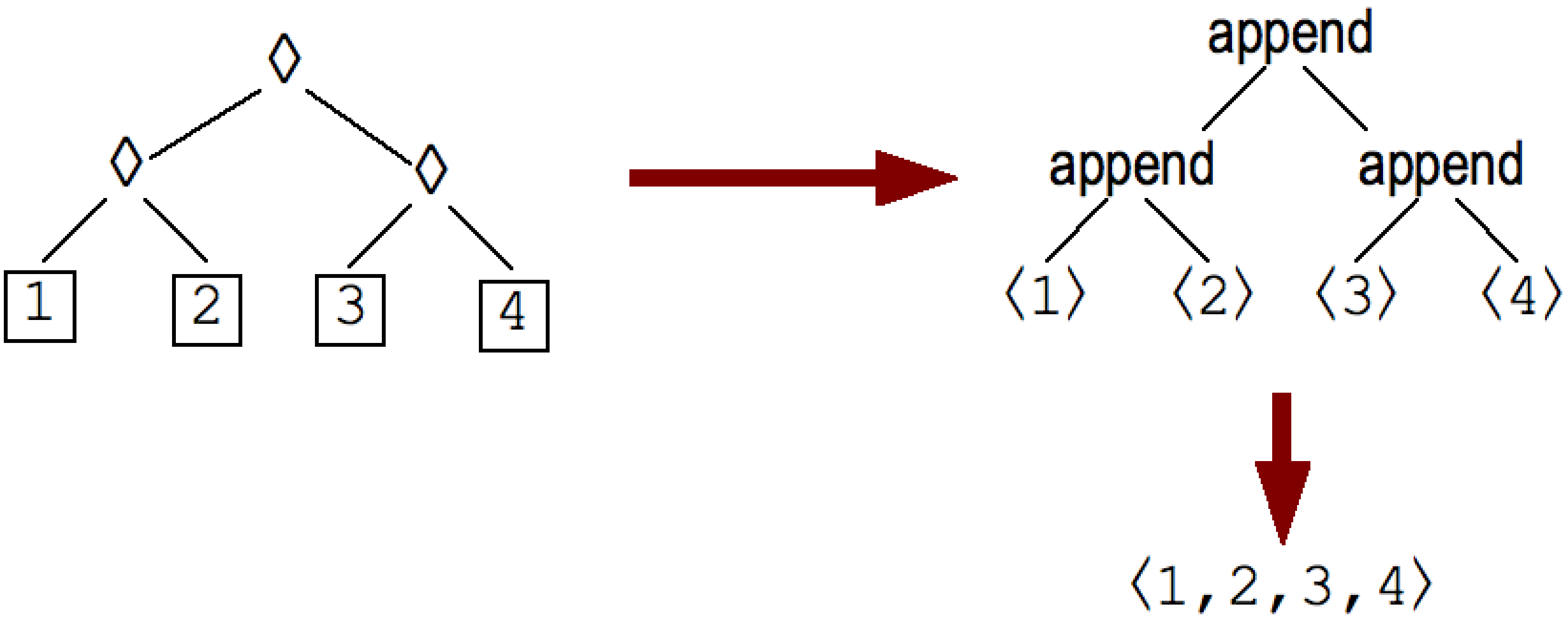
Computation: Lists

Replace $\Diamond$ $\Box$ ε with append $\langle - \rangle$ $\langle\rangle$



Representation: Lists

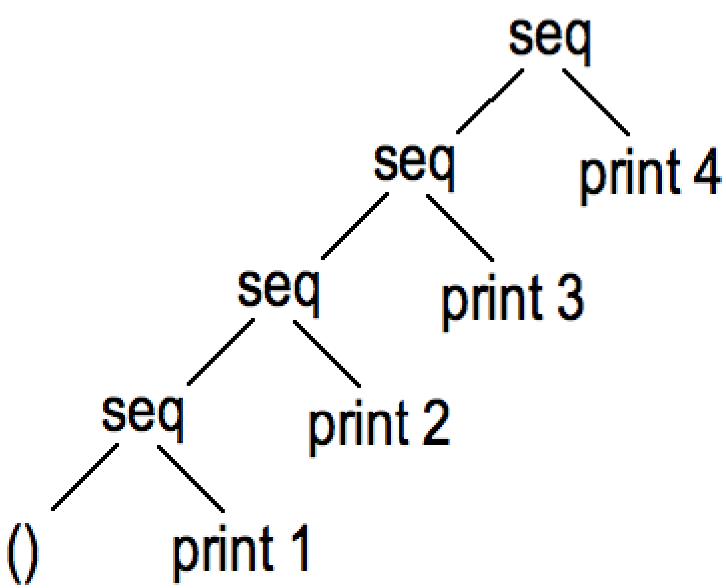
Replace $\Diamond$ $\Box$ ε with append $\langle - \rangle$ $\langle\rangle$

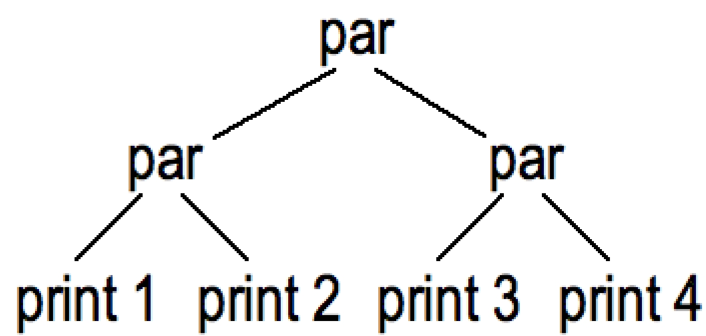


Catamorphism: Loops

Replace ◊ □ ε with seq identity () or par identity ()
where seq: (),() → () and par: (),() → ()

for $i \leftarrow seq(1:4)$ do $print\ i$ end





for $i \leftarrow 1:4$ do $print\ i$ end

Generators can modify the catamorphism
and so control the parallelism.

To Summarize: A Big Idea

- Loops and summations and list constructors are alike!

$$\texttt{for}\ i \leftarrow 1\!:\!1000000\ \texttt{do}\ x_{\mathrm{i}} := x_{\mathrm{i}}^2\ \texttt{end}$$

$$\sum_{i \leftarrow 1:1000000} x_{\mathrm{i}}^2$$

$$\langle\, x_{\mathrm{i}}^2 \mid i \leftarrow 1\!:\!1000000 \,\rangle$$

  > Generate an abstract collection
  > The *body* computes a function of each item
  > Combine the results (or just synchronize)

- Whether to be sequential or parallel is a separable question
  > That's why they are especially good abstractions!
  > Make the decision on the fly, to use available resources

Another Big Idea

- Formulate a sequential loop as successive applications of state transformation functions f_i
- Find an *efficient* way to compute and represent compositions of such functions (this step requires ingenuity)
- Instead of computing

  $s := s_0; \texttt{for}\ \ i \leftarrow seq(1:1000000)\ \texttt{do}\ \ s := f_i(s)\ \texttt{end}$,

  compute $s := (\bigcirc_{i \leftarrow 1:1000000} f_i)\ s_0$
- Because function composition is associative, the latter has a parallel strategy
- In the "words in a string" problem, each character can be regarded as defining a state transformation function

We Need a New Mindset

- DO loops are so 1950s!
- So are linear linked lists!
- Java™-style iterators are **so** last millennium!
- Even arrays are suspect!
- As soon as you say "first, `SUM = 0`" you are hosed. Accumulators are BAD.
- If you say, "process subproblems in order," you lose.
- The great tricks of the sequential past DON'T WORK.
- The programming idioms that have become second nature to us as everyday tools DON'T WORK.

Fortress: A Parallel Language

High productivity for multicore, SMP, and cluster computing

- Hard to write a program that isn't potentially parallel
- Support for parallelism at several levels
  - Expressions
  - Loops, reductions, and comprehensions
  - Parallel code regions
  - Explicit multithreading
- Shared global address space model with shared data
- Thread synchronization through atomic blocks and transactional memory

These Are All Potentially Parallel

$$f(a) + g(b)$$

$$L = \langle\, \mathit{find}(k, x) \mid k \leftarrow 1\!:\!n, x \leftarrow A \,\rangle$$

$$s = \sum_{k \leftarrow 1:n} c_k\, x^k$$

```
for k ← 1:n do
    a_k := b_k
    sum += c_k x^k
end
```

$\texttt{for}\ \ k \leftarrow 1\!:\!n\ \texttt{do}$
$\quad a_k := b_k$
$\quad \mathit{sum} \mathrel{+}= c_k\, x^k$
$\texttt{end}$

$\texttt{do}$
$\quad f(a)$
$\texttt{also do}$
$\quad g(b)$
$\texttt{end}$

$\texttt{do}$
$\quad T_1 = \texttt{spawn}\ \ f(a)$
$\quad T_2 = \texttt{spawn}\ \ g(b)$
$\quad T_1.\mathit{wait}(); T_2.\mathit{wait}()$
$\texttt{end}$

Mathematical Syntax 1

Integrated mathematical and object-oriented notation

- Supports a stylistic spectrum that runs from Fortran to Java™—and sticks out at both ends!
  - > More conventionally mathematical than Fortran
    - – Compare `a*x**2+b*x+c` and $a\,x^2 + b\,x + c$
  - > More object-oriented than Java
    - – Multiple inheritance
    - – Numbers, booleans, and characters are objects
  - > To find the size of a set S: either $|S|$ or $S.size$
    - – If you prefer $\#\,S$, defining it is a one-liner.

Mathematical Syntax 2

- Full Unicode character set available for use, including mathematical operators and Greek letters:

$\times$	$\div$	$\oplus$	$\ominus$	$\otimes$	$\oslash$	$\odot$	$\approx$	α	β	γ	δ
$\boxplus$	$\boxminus$	$\boxtimes$	$\leftrightarrow$	$\wedge$	$\vee$	$\equiv$	$\not\equiv$	ϵ	ζ	η	θ
$\leq$	$\geq$	$\sum$	$\prod$	$\prec$	$\preccurlyeq$	$\succcurlyeq$	$\succ$	ι	κ	λ	μ
$\cap$	$\cup$	$\uplus$	$\subset$	$\subseteq$	$\supseteq$	$\supset$	$\in$	ξ	π	ρ	σ
$\sqcap$	$\sqcup$	$\sqsubset$	$\sqsubseteq$	$\sqsupseteq$	$\sqsupset$	$\neg$	$\notin$	ϕ	χ	ψ	ω
$\lfloor$	$\rfloor$	$\lceil$	$\rceil$	$\langle$	$\rangle$	$\curlywedge$	$\curlyvee$	Γ	Θ	and so on	

- Use of “funny characters” is under the control of libraries (and therefore users)

Visit http://projectfortress.sun.com

An open-source project with international participation

- Open source since January 2007
- University participation includes:
  - University of Tokyo: matrix algorithms
  - Rice University: code optimization
  - Aarhus University: syntactic abstraction
  - University of Texas at Austin: static type checking
- Also participation by many individuals

A Growing Library

The Fortress library now includes over 12,000 lines of code.

- Integer, floating-point, and string operations
- Big integers, rational numbers, intervals
- Collections (lists, sets, maps, heaps, etc.)
- Multidimensional arrays
- Sparse vectors and matrices
- Generators and reducers
  - Implement loops, comprehensions, and reductions
  - Support implicit parallelism
- Fortress abstract syntax trees
- Sorting

Tools: 'Fortify' Code Formatter

- Emacs-based tool
- Fortress programs can be typed on ASCII keyboards
- Code automatically formatted for processing by $\LaTeX$

```
sum: RR64 := 0
for k<-1:n do
  a[k] := (1-alpha)b[k]
  sum += c[k] x^k
end
```

$$
\begin{array}{l}
sum\colon \mathbb{R}64 := 0 \\
\texttt{for}\ k \leftarrow 1\!:\!n\ \texttt{do} \\
\quad a_k := (1-\alpha) b_k \\
\quad sum \mathrel{+}= c_k\, x^k \\
\texttt{end}
\end{array}
$$

All code on these slides was formatted by this tool.

Tools: Editing Environments

- Fortress mode for Emacs
  - Provides syntax coloring
  - Some automatic formatting
  - Unicode font conversion
- Fortress NetBeans™ plug-in
  - Syntax highlighting
  - Mark occurrences
  - Instant rename
- These tools were contributed by people outside Sun

Syntax Coloring Screen Shot

quicksort

```
(* Quicksort *)

quicksort⟦T⟧(lt:(T,T)→Boolean, arr:Array⟦T,ℤ32⟧, left:ℤ32, right:ℤ32) =
  if right > left
  then pivotIndex = left
       pivotNewIndex = mypartition(lt, arr, left, right, pivotIndex)
       do
         quicksort(lt, arr, left, pivotNewIndex−1)
       also do
         quicksort(lt, arr, pivotNewIndex+1, right)
       end
  end
```

1:-- **quicksort** All (3,0) (Fortress)

Wrote /Users/gls/quicksort

Fortress 1.0

- With the Fortress 1.0 release in March 2008, we synchronized the specification and implementation
- Implementation expanded and made more reliable since Fortress 1.0β
- Many features in the 1.0β specification were removed for 1.0
  - *But with every intention of adding them back as the language grows*
  - And we have done so over the last six months

What works NOW

- Parallelism in loops, reductions, comprehensions, tuples
- Automatic load balancing via work-stealing

$$
\begin{array}{l}
\texttt{for}\ \ i \leftarrow 0\ \texttt{\#}\ |\mathit{children}'|\ \texttt{do} \\
\quad \mathit{children}'_i := \mathit{generate\_tail}[\![\mathrm{Key}, \mathrm{Val}]\!](\mathit{children}_{i+lsize+1}, 1) \\
\texttt{end}
\end{array}
$$

$$\mathit{factorial}(n\colon \mathbb{Z}32) = \prod_{i \leftarrow 1:n} i$$

$$\texttt{opr}\ (n\colon \mathbb{Z}32)! = \prod_{i \leftarrow 1:n} i$$

$$\langle\, x^2 \mid x \leftarrow \{0, 1, 2, 3, 4, 5\}, x\ \texttt{MOD}\ 2 = 0 \rangle$$

What works NOW

- Spawn

```
spawn do
```

$$s := \mathrm{Done}[\![T]\!](old.val())$$

```
end
```

What works NOW

- Atomic blocks with transactional memory

$$
\begin{array}{l}
\mathit{attempt}()\colon (\mathrm{State}[\![T]\!], \mathrm{Boolean}) = \texttt{atomic do} \\
\qquad \mathit{old} = s \\
\qquad \mathit{computed} := \mathit{old}.\mathit{isDone}() \\
\qquad \texttt{if } \neg \mathit{old}.\mathit{isDone}() \texttt{ then} \\
\qquad\qquad \texttt{if } \mathit{old}.\mathit{isPending}() \texttt{ then } \mathit{abort}() \texttt{ end} \\
\qquad\qquad s := \mathrm{Pending}[\![T]\!] \\
\qquad\qquad (\mathit{old}, \mathit{true}) \\
\qquad \texttt{else} \\
\qquad\qquad (\mathit{old}, \mathit{false}) \\
\qquad \texttt{end} \\
\texttt{end}
\end{array}
$$

What works NOW

- Object-oriented type system with multiple inheritance
- Overloaded methods and operators with dynamic multimethod dispatch
- Sets, arrays, lists, maps, skip lists
- Pure queues, deques, priority queues
- Integers, floating-point, strings, booleans
- Big integers, rational numbers, interval arithmetic
- Syntactic abstraction (just barely)

Next steps:

- Full static type checker (almost there!)
- Static type inference to reduce “visual clutter”
- Parallel nested transactions
- Compiler
  - > Initially targeted to JVM for full multithreaded platform independence
  - > After that, VM customization for Fortress-specific optimizations

The Parallel Future

- We need to teach new strategies for problem decomposition.
  - Data structure design/object relationships
  - Algorithmic organization
  - Don't split a problem into "the first" and "the rest."
  - Do split a problem into roughly equal pieces.
    Then figure out how to combine general subsolutions.
  - Often this makes combining the results a bit harder.
- We need programming languages and runtime implementations that support parallel strategies *and* hybrid sequential/parallel strategies.
- We must learn to manage new space-time tradeoffs.

Conclusion

- A program organized according to linear problem decomposition principles can be really hard to parallelize.
- A program organized according to parallel problem decomposition principles is easily run either in parallel or sequentially, according to available resources.
- The new strategy has costs and overheads. They will be reduced over time but will not disappear.
- This is our only hope for program portability in the future.

It is an exciting time for the project

- External contributions and feedback are increasing
  - Thank you!
- Many implementation tasks are being done outside Sun
- The language is growing
- A community of developers is participating in its evolution

guy.steele@sun.com
http://research.sun.com/projects/plrg
http://projectfortress.sun.com