

Parallel Computing in Fortress (Part II)

Jan-Willem Maessen

Programming Language Research Group
Sun Microsystems Laboratories
Burlington, MA, USA

TiC 2008, 24 June 2008, Prague

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.

Advertisement

- The Strahov monastery in Hradčany, above the castle, has a brewery restaurant
- It's very nice, and my wife (who doesn't drink) wants to go back
- It's about 30-40min on "scenic trams 22 or 23" from here
- If you're interested in joining us, talk to me
- Czechs: how do you buy tram tickets in town without going to a Metro station?

Generators and Reductions

Generator $[[T]] \Rightarrow$ Reduction $[[T]]$

Generator $[[T]]$ | Reduction $[[T]]$

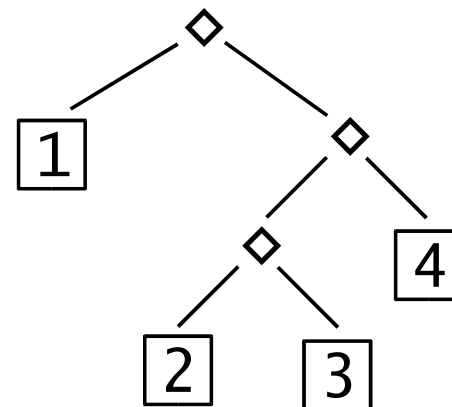
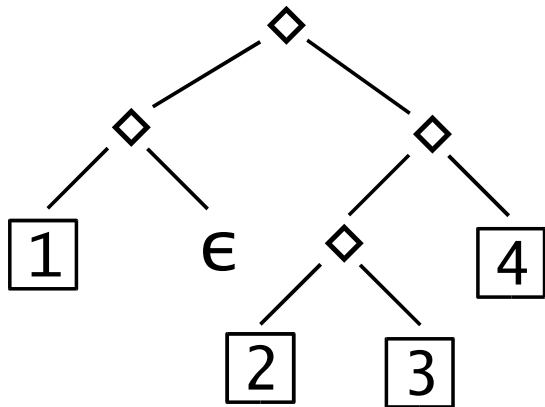
- Idea: Express all iteration in Fortress as library code
- Fuse generators and reductions to yield efficient code
- Exploit properties of each to specialize the code

Representation of abstract collections

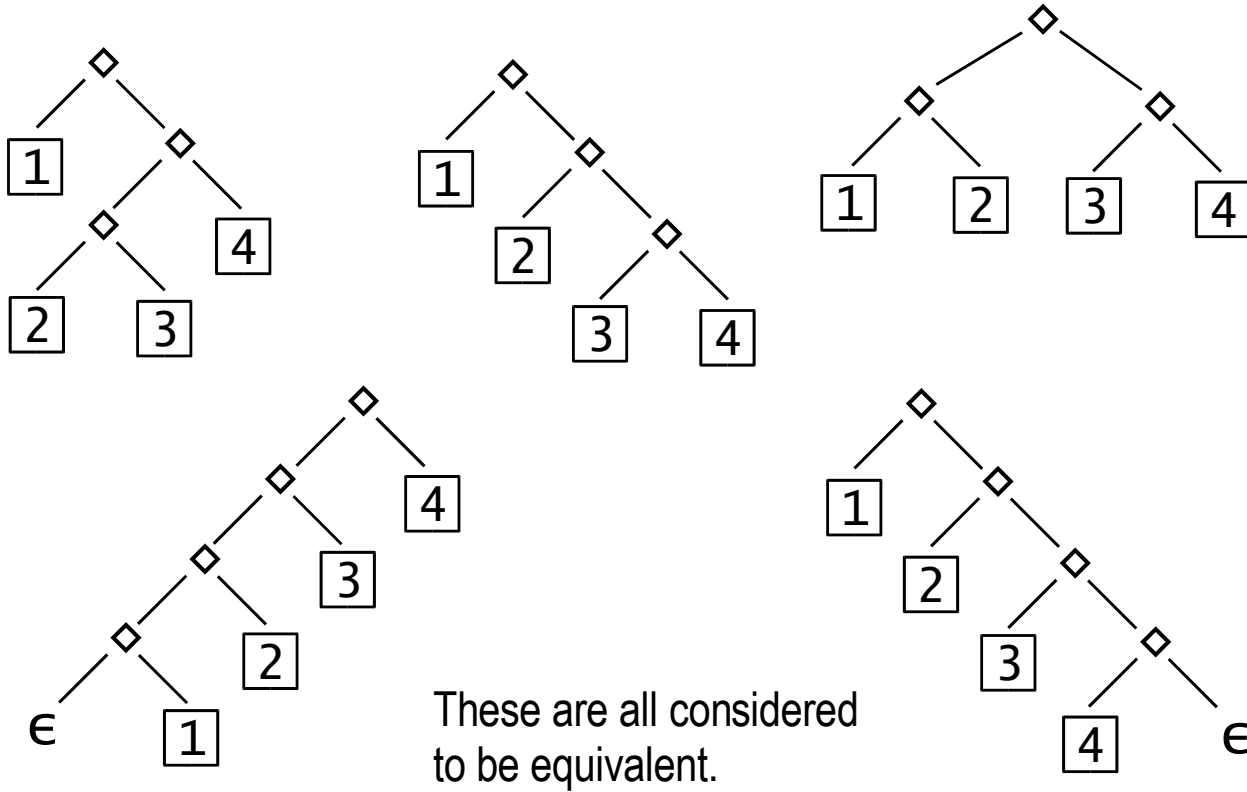
Binary operator \diamond

Leaf operator (“unit”) \square

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

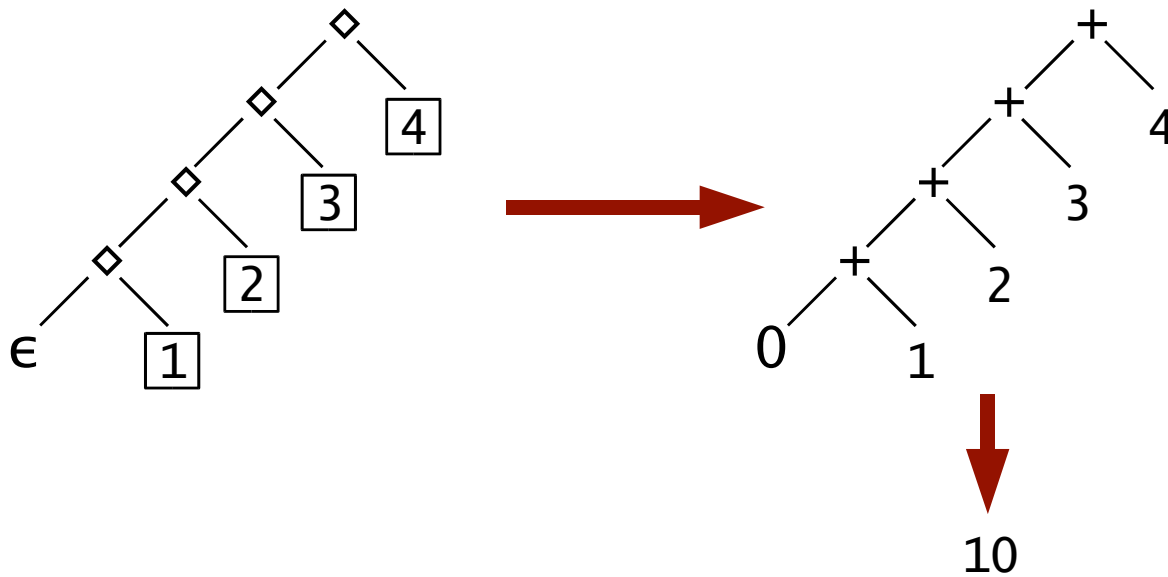


Abstract collections are monoids



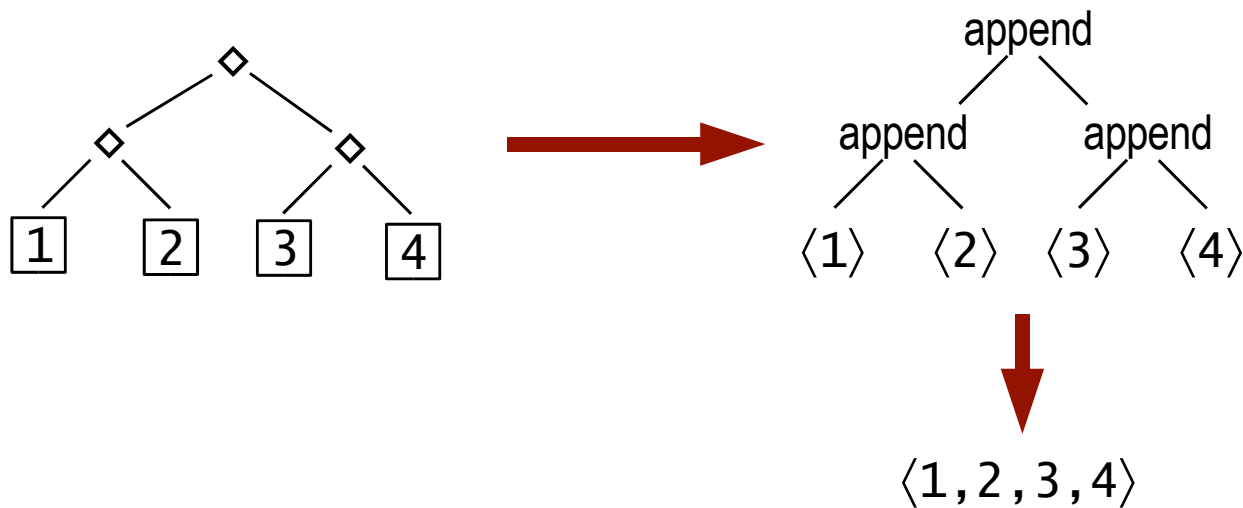
Abstract traversal = Catamorphism

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



Catamorphism for List

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



The Reduction type

```
trait Reduction[[L]]  
  empty(): L  
  join(a: L, b: L): L  
end
```

- Associative operation *join* with identity *empty*
- Note that this does not form a complete catamorphism!

Actual Reductions

```
trait ActualReduction[[R, L]] extends Reduction[[L]]
  lift(r: Any): L
  unlift(l: L): R
end
```

```
trait MonoidReduction[[R]] extends ActualReduction[[R, R]]
  lift(r: Any): R = r
  unlift(r: R): R = r
end
```

```
trait CommutativeMonoidReduction[[R]] extends MonoidReduction[[R]]
end
```

- Lift and unlift allow us to impose an identity if necessary.

Concatenation reduction

```
object Concat[[T]] extends MonoidReduction[[List[[T]]]]
  empty(): List[[T]] = Empty[[T]]
  join(a: List[[T]], b: List[[T]]) = a APP b
end
```

```
opr BIG APP [[T]](): BigReduction[[List[[T]], List[[T]]] =
  BigReduction(Concat[[T]])
```

- We return a BigReduction.
- Packages up ActualReduction for use in desugaring.

Defining new Generators

- Approach 1: Define in terms of existing generators
 - > Advantage: simple (as we shall see).
 - > Disadvantage: inflexible. Must follow generator's layout and protocol.
- Approach 2: Write from scratch
 - > Advantage: Flexible. Can work any way we want.
 - > Advantage: Only necessary to define one method
 - > Disadvantage: Usually a good idea to define several other methods, and a helper type for *seq*.

Deriving one Generator from another

```
sieve() : () = do
  prime[0] := false
  prime[1] := false
  for (p, isPrime) ← seq(prime.indexValuePairs()),
    isPrime do
    removeMultiplesOf(p)
  end
end

primeString() : String =
  BIG|||[ (p, isPrime) ← prime.indexValuePairs(),
    isPrime] p
```

Deriving one Generator from another

```
primes : Generator[[Z32]] = do
  ivs: Generator[[ (Z32, Boolean) ]] =
    prime.indexValuePairs()
  isPrime(_, p : Boolean): Boolean = p
  primeIvs : Generator[[ (Z32, Boolean) ]] =
    ivs.filter(isPrime)
  primeIvs.map[[Z32]](fn (p : Z32, _): Z32 ⇒ p)
end
```

- Doesn't actually do any enumeration
- Simply creates new objects of type Generator

Using our custom generator

```
sieve(): () = do  
  prime[0] := false  
  prime[1] := false  
  for p ← primes do  
    removeMultiplesOf(p)  
  end  
end
```

```
primeString(): String =  
  BIG ||| [p ← primes] p
```

Making the code even more concise

```
sieve(): () = do  
    prime[0] := false  
    prime[1] := false  
    removeMultiplesOf(p), p ← primes  
end  
  
primeString(): String = BIG ||| primes
```

The type Generator

```
trait Generator[[E]]
  abstract generate[[R]](r: Reduction[[R]], body: E → R): R
  seq(self): SequentialGenerator[[E]]
  map[[G]](f: E → G): Generator[[G]]
  nest[[G]](f: E → Generator[[G]]): Generator[[G]]
  filter(f: E → Condition[()]): Generator[[E]]
  cross[[G]](g: Generator[[G]]): Generator[[E, G]]
  mapReduce[[R]](body: E → R, join: (R, R) → R, zero: R): R
  reduce(j: (E, E) → E, z: E): E
  reduce(r: Reduction[[E]]): E
  loop(f: E → ()): ()
  opr ∈(x: E, self): Boolean
end
```

Making List into a Generator

```
trait List[[T]] extends { Generator[[T]] }
  comprises { Empty[[T]], NonEmpty[[T]] }
  generate[[R]](red: Reduction[[R]], body : T → R) : R
end

object Empty[[T]] extends List[[T]]
  generate[[R]](red: Reduction[[R]], body : T → R) : R =
    red.empty()
end

object Singleton[[T]](t : T) extends NonEmpty[[T]]
  generate[[R]](red: Reduction[[R]], body : T → R) : R =
    body(t)
end
```

Making List into a Generator: Append

```
object Append[[T]](l: NonEmpty[[T]], r: NonEmpty[[T]])
  extends NonEmpty[[T]]
  generate[[R]](red: Reduction[[R]], body: T → R): R =
    red.join(l.generate[[R]](red, body),
             r.generate[[R]](red, body))
end
```

- Note where the parallelism is coming from!

Defining seq

```
trait List[[T]] extends { Generator[[T]] }  
  comprises { Empty[[T]], NonEmpty[[T]] }  
  seq(self): SeqList[[T]] = SeqList[[T]](self)  
  seqgen[[R]](red: Reduction[[R]], body : T → R) : R  
end  
  
object SeqList[[T]](xs : List[[T]]) extends SequentialGenerator[[T]]  
  generate[[R]](red: Reduction[[R]], body : T → R) : R =  
    xs.seqgen[[R]](red, body)  
end
```

- Idiom: Define a wrapper type
- Add an extra method for sequential generation
- Wrapper calls through to this method

Defining the *seqgen* method

```
object Append[[T]](l : NonEmpty[[T]], r : NonEmpty[[T]])  
  extends NonEmpty[[T]]  
  seqgen[[R]](red: Reduction[[R]], body : T → R) : R = do  
    ll = l.seqgen[[R]](red, body)  
    rr = r.seqgen[[R]](red, body)  
    red.join(ll, rr)  
  end  
end
```

- Note the use of binding to sequence computations!
- For Empty and Singleton: same as *generate*

Desugaring Generated Expressions

```
for gs do
  exprs
end
e, gs
 $\langle e \mid gs \rangle$ 
BIG  $\oplus [gs] e$ 
```

Three pieces of every generated expression:

- Generator list
- body
- Operation combining body results into final result

Desugaring Generated Expressions

for *gs* do
 exprs
 end \Rightarrow $bigOperator(\text{Loop}, (\text{fn } (reduction, body) \Rightarrow DS[\text{do } exprs \text{ end} | gs]))$

e, gs \Rightarrow $bigOperator(\text{Loop}, (\text{fn } (reduction, body) \Rightarrow DS[e | gs]))$

$\langle e | gs \rangle$ \Rightarrow $bigOperator(\text{BIG } \langle \rangle, (\text{fn } (reduction, body) \Rightarrow DS[e | gs]))$

$\text{BIG } \oplus [gs] e$ \Rightarrow $bigOperator(\text{BIG } \oplus (), (\text{fn } (reduction, body) \Rightarrow DS[e | gs]))$

The bigOperator function

```
trait BigOperator[[I, O, R, L]]
  getter reduction(): ActualReduction[[R, L]]
  getter body(): I → R
  getter unwrap(): R → O
end

bigOperator[[I, O, R, L]](
  o: BigOperator[[I, O, R, L]],
  desugaredClauses : (Reduction[[L], I → L) → L): O = do
  r = o.reduction()
  body(i): L = r.lift((o.body()))(i)
  (o.unwrap())(r.unlift(desugaredClauses(r, body)))
end
```

Desugaring Generator Lists Recursively

$$\text{BIG } \oplus [gs] e \quad \Rightarrow \quad \text{bigOperator}(\text{BIG } \oplus (), \\ (\text{fn } (reduction, body) \Rightarrow \text{DS}[[e | gs]]))$$

$$\text{DS}[[e |]] \quad \Rightarrow \quad \text{body}(e)$$

$$\text{DS}[[e | \vec{x} \leftarrow g, gs]] \quad \Rightarrow \quad (g).\text{generate}(reduction, \\ \text{fn } \vec{x} \Rightarrow \text{DS}[[e | gs]])$$

$$\text{DS}[[e | p, gs]] \quad \Rightarrow \quad (p).\text{generate}(reduction, \\ \text{fn } () \Rightarrow \text{DS}[[e | gs]])$$

A more parallel sieve

Make the prime sieve handle as many candidate primes as possible in parallel.

- First composite divisible only by p_i is p_i^2 .
- Once we've crossed out all multiples of p_i , any number between p_i^2 and p_{i+1}^2 that looks prime, is prime.
- Process all the primes between adjacent prime squares in parallel.

A more parallel sieve: solution

```
sieve(): () = do
  prime[0] := false
  prime[1] := false
  prevSquared : Z32 := 1
  for p ← seq(2: [√upper]), prime[p] do
    pSquared = p2
    for p' ← (prevSquared + 1) : (pSquared - 1), prime[p'] do
      removeMultiplesOf(p')
    end
    prevSquared := pSquared
  end
end
```

Histogramming final digits

- Question: How many primes end with each digit?

```
hist():  $\mathbb{Z}32[10]$  = do  
  result = array1[[ $\mathbb{Z}32$ , 10]](0)  
  for p ← primes do  
    lastDigit = p MOD 10  
    atomic result[lastDigit] += 1  
  end  
  result  
end
```

```
[0#10] = [ 0 40 1 42 0 1 0 46 0 38 ]
```

Transactional Atomicity

- All actions within an atomic region should appear to occur atomically.
- Should not be able to observe the outside world changing from inside the atomic region.
- The outside world should not observe intermediate states of the running transaction.
- The transaction is serializable with respect to the outside world—all its reads and write appear to occur at a single point in time.
- Atomicity applies to *all* accesses to memory, not just other atomic regions (ie Fortress is strongly atomic).

Transactional Atomicity in Fortress

- Goal: composability. Should be possible to combine two arbitrary pieces of code together into an atomic action.
- Let the run time deal with the necessary locking and synchronization.
- Consequence: within an atomic block the “whole” language is available to us.
 - > Nested atomic blocks
 - > Parallelism
 - > Data races!
- However, there are things we can't figure out how to do:
 - > Interact irrevocably with the outside world
 - > Start or stop explicit threads...

The Bank Account Example

```
object Account(var balance :  $\mathbb{Z}32$ )
  deposit(amount:  $\mathbb{Z}32$ ): () = atomic do
    balance += amount
  end
  withdraw(amount:  $\mathbb{Z}32$ ): () = atomic do
    balance -= amount
  end
end

transfer(amount:  $\mathbb{Z}32$ , fromAcct: Account, toAcct: Account): () =
  atomic do
    (fromAcct.withdraw(amount), toAcct.deposit(amount))
  ()
end
```

Transactional Pragmatics

- Could just take a big global lock, but that isn't parallel!
- Instead, use fine-grained read/write locking, conceptually on individual memory locations.
- Goal: if two transactions touch disjoint data, they run in parallel.
- I'm going to skim over a lot of details about when and how the fine-grained locking occurs; our semantics don't tell us what choices to make.

Parallel Transactions can Fail

- Don't necessarily know in advance what locations a given atomic block will access (data-dependent).
- Locking resources in an arbitrary order \Rightarrow deadlock!
- When we attempt to access a resource already being accessed by another thread, and one of the accesses is a write, we say a **conflict** has occurred.
- Conflict may require us to **abort** one of the conflicting atomic blocks and roll it back.
- Thus an atomic block must keep track of any changes it makes (both to global state and to local variables), so that it can discard those changes if necessary.

Abrupt Termination

```
object Account(var balance : Z32)
  deposit(amount: Z32): () = atomic do
    balance += amount
  end
  withdraw(amount: Z32): () throws Overdrawn = atomic
    if amount ≤ balance then
      balance -= amount
    else
      throw Overdrawn(balance - amount)
    end
  end
end

transfer(amount: Z32, fromAcct: Account, toAcct: Account): () throws Overdrawn =
  atomic do
    (fromAcct.withdraw(amount), toAcct.deposit(amount))
  ()
end
```

Exception semantics

- What happens when execution of the body of `atomic` terminates abruptly?
 - > In Fortress, we consider the atomic expression to have aborted, and roll back its effects.
 - > Except effects on newly-allocated objects (such as any exception object we might be throwing).
 - > Similar to the model used by Haskell's transactional atomicity.
- Note that this commits us to an implementation based on rollback.

Boosting: Method-level Atomicity

- What I've described so far is [transactional memory](#).
- Units of synchronization: individual reads and writes
- Idea: [Transactional Boosting](#) (Herlihy & Koskinen): package up a data structure so that units of atomicity are method calls.
- We'd have to figure out how those method calls might conflict with one another, and provide some way of undoing them in case of conflict.
- We have a disciplined model of boosting based on open nesting, but we haven't seen enough semantics yet...

Tryatomic and abort

```
opr ORELSE[[T]](block1 : () → T, block2 : () → T) =
  try
    tryatomic block1()
  catch x
    TryAtomicFailure ⇒
      block2() : ORELSE: block1()
  end
```

```
withdrawalWithOverdraftProtection(amt : ℤ32, account : Account) =
  (if account.balance < amt then
    abort
  else
    account.withDraw(amt)
  end: ORELSE :
  if account.balance ≥ amt then
    abort
  else
    obtainLoan(amt - account.balance)
    account.withDraw(account.balance)
  end)
```

Lack of fairness

```
lazyData : Maybe[[Z32]] := Nothing[[T]]

computeLazyData(): () = do
  lazyData := Just(42)
end

getLazyData(): Z32 = atomic
  if d ← lazyData then
    d
  else
    abort
  end

do
  computeLazyData()
also do
  result := getLazyData()
end
```

- Using *abort*, it is possible for a thread to block / spin.
- That usually means we're waiting for something to happen.
- Implicit threads in Fortress are **unfair!**
- So this doesn't work.

Spawned threads are fair

```
lazyData : Thread[[Z32]] :=  
  spawn computeLazyData()  
computeLazyData(): Z32 = 42  
getLazyData(): Z32 = lazyData.val()  
do  
  result := getLazyData()  
end
```

- The spawn construct returns a future.
- Every implicit (unfair) thread belongs to some spawned (fair) thread.
- Computations belonging to disjoint spawned threads are scheduled fairly with respect to one another.

Operational semantics: a beginning

Expression	$E ::= x \mid ()$ $\mid \vec{E} \mid E(E)$ $\mid \text{fn } \vec{x} \Rightarrow E$ $\mid \text{Ref}(E) \mid E.\text{val}$ $\mid \text{do } \vec{A}; E \text{ end}$ $\mid \text{atomic do } \vec{A}; E \text{ end}$ $\mid \text{do } \vec{A}; E \text{ end: ORIG:}$ $\mid \text{do } \vec{A}; E \text{ end}$	Value	$V ::= \vec{S} \mid ()$ $\mid \text{fn } \vec{x} \Rightarrow E$
		Simple term	$S ::= x \mid V$
		Heap term	$H ::= x = S$ $\mid x = \text{Ref}(S)$
Action	$A ::= \vec{x} = E$ $\mid E.\text{val} := E$ $\mid E.\text{update}(S, S)$ $\mid E$	Pending	$P ::= x.\text{update}(S, S)$
		Completed	$C ::= H \mid P$

- Simple binding-oriented calculus
- Initial goal: just capture the parallelism behavior.

Finger exercises

$(\text{fn } \vec{x} \Rightarrow e)(e')$	$\rightarrow \text{do } \vec{x} = e'; e \text{ end}$	β_{let}
$\text{do } C; x = S; A_I[x] \text{ end}$	$\rightarrow \text{do } C; x = S; A_I[S] \text{ end}$	Instantiation
$(x_1, x_2, \dots, x_n) = (s_1, s_2, \dots, s_n)$	$\rightarrow x_1 = s_1; x_2 = s_2; \dots, x_n = s_n$	Tuple destructuring
$E_L[\text{do } C; S \text{ end}]$	$\rightarrow \text{do } C; E_L[S] \text{ end}$	Do hoisting
$\vec{x} = \text{do } C; S \text{ end}$	$\rightarrow C; \vec{x} = S$	Flattening (1)
$\text{do } C; \text{do } C'; S \text{ end} \text{ end}$	$\rightarrow \text{do } C; C'; S \text{ end}$	Flattening (2)

- Every binding-based small-step semantics has similar rules
- *Very* small-step semantics (eg one instantiation at a time)
- Goal: actually see parallelism and atomicity in action
- Note that we never instantiate $x = \text{Ref}(S)$.
- We haven't said anything about *evaluation strategy* yet.

Strategy

$$\begin{aligned}
 E_R[] ::= & [] \\
 & | (\vec{E}, E_R[], \vec{E}) \\
 & | E_R[](E) \\
 & | E(E_R[]) \\
 & | \text{Ref}(E_R[]) \mid E_R[].val \\
 & | \text{do } C; \vec{x} = E_R[]; E \text{ end} \\
 & | \text{do } C; E_R[] \text{ end} \\
 & | E_R[] : \text{ORIG}: \text{do } \vec{A}; E \text{ end}
 \end{aligned}$$

- Also, instantiation contexts = reduction contexts
- Parallelism = non-unique choice of redex
- Non-fairness = no limit on how often redex chosen

Representing transactions

- Can be multiple copies of a single variable
- Each atomic section tracks data it accessed
- This is tree-like and open-ended
- So: **embed heap information in the program term!**
- Each atomic section tracks its memory accesses
- Need to keep old and new memory contents around

Beginning a transaction

`atomic do A; E end` \rightarrow

`(do A; E end) : ORIG: (do A; E end)`

- Keep original atomic block in case of abort

Fetch from memory

do $C; x = \text{Ref}(S); A_F[x.val]$ end \rightarrow do $C; x = \text{Ref}(S); A_F[S]$ end

do $C; x.update(S_1, S_2); A_F[x.val]$ end \rightarrow do $C; x.update(S_1, S_2); A_F[S_2]$ end

do $C; x = \text{Ref}(S); A_H[\text{do } C'; A_F[x.val] \text{ end: ORIG: } E]$ end \rightarrow

do $C; x = \text{Ref}(S); A_H[\text{do } C'; x.update(S, S); A_F[S] \text{ end: ORIG: } E]$ end

do $C; x.update(S_1, S_2); A_H[\text{do } C'; A_F[x.val] \text{ end: ORIG: } E]$ end \rightarrow

do $C; x.update(S_1, S_2); A_H[\text{do } C'; x.update(S_2, S_2); A_F[S_2] \text{ end: ORIG: } E]$ end

- Here $A_F[\]$ is an immediate memory context
 - > No intervening atomic blocks.
- By contrast, $A_H[\]$ is a nested memory context
 - > Any # of atomics, no mention of x
- Meaning of $x.update(S_0, S_1)$: saw S_0 , updated to S_1

Conflict

do $C; x = \text{Ref}(S_0); A_H[\text{do } C; x.\text{update}(S_2, S_3); A \text{ end: ORIG: } E]$
→ do $C; x = \text{Ref}(S_0); A_H[E : \text{ORIG: } E]$ end

do $C; x = \text{update}(S_0, S_1); A_H[\text{do } C; x.\text{update}(S_2, S_3); A \text{ end: ORI}$
→ do $C; x = \text{update}(S_0, S_1); A_H[E : \text{ORIG: } E]$ end

- Here $S_0 \neq S_2$
- We throw out the partially-executed block and retry it
- Doing so throws away all the modifications to the heap

Commit

do C_0 ; A_F [do C_1 ; S end: ORIG: E] end \rightarrow
 do $merge(C_0, C_1)$; A_F [S] end

- Only applies if there is NO conflict redex in this sub-term.
- The merge operation is specified as follows for overlaps:
 - > $merge(x = \text{Ref}(S_0), x.update(S_0, S_1)) = x = \text{Ref}(S_1)$
 - > $merge(x.update(S_0, S_1), x.update(S_1, S_2)) = x.update(S_0, S_2)$
- Non-overlapping modifications and value bindings are simply preserved.

A Work in Progress

- Pretty sure we haven't quite gotten the rules for transactions right
- Didn't show rules for throw, but they're pretty easy
- Abort should be simple
- Make reduction rules less strategy-dependent
- Make sure semantics matches implementation
- Show that we can serialize implicit threads
- Understand when abort causes deadlock/livelock
- Add fair threads (makes strategy fiddly)
- Add the memory model (upheaval!)

The State of Fortress

- We have a usable interpreter written in Java
- It runs stuff in parallel, even nested transactions
- Work has begun on the compiler
- Syntax abstraction prototyped, final version nearly done
- Group at U. Tokyo working on nested generators:
 - > Example: heads and tails
 - > Fuse for efficient traversal
 - > Turn into parallel prefix if not fusible
- As each new piece comes on line, library upheaval!
- Now easy to dash off a quick fortress program like *sieve*

Exercises

- Define an operator that takes two (index,value) pairs and returns the pair with maximum value.
- Using that, define a corresponding reduction and big operator.
- Design an operator $a ||| b$ that takes two generators a and b and returns a generator yielding the elements of a and b in parallel, but with the elements of a occurring before the elements of b in natural order. Don't forget *seq!*
- Write a parallel sort on lists.

Answers? Questions? JanWillem.Maessen@sun.com

Where to get Fortress

`http://projectfortress.sun.com/`

Subversion download gives the latest version of the code:

```
svn checkout https://projectfortress.sun.com/svn/Community/trunk PFC
```

Questions about this tutorial? JanWillem.Maessen@sun.com

<http://projectfortress.sun.com/>

