Quicksort, again

David Kitchin, Adrian Quark, Jayadev Misra

Department of Computer Science
University of Texas at Austin

http://orc.csres.utexas.edu
Describe Quicksort using:

- **Recursion**: nicely done in Functional programming
- **Mutable Data Structure**: nicely done in Imperative programming
- **Concurrency**: ?
Orc, an Orchestration Theory

- **Site**: Basic service (component).
- **Concurrency** combinators for integrating sites.
- **Theory** includes nothing other than the combinators.

No notion of data type, thread, process, channel, synchronization, ⋯
Sites

- A site is called like a procedure with parameters.
- Site returns at most one value.
- The value is *published*.

Site calls are *strict*. 
Examples of Sites

- + - * && || < = ...
- println, random, Prompt, Email ...
- Ref, Semaphore, Channel, Database ...
- Timer

- **External Services:** Google Search, MySpace, CNN, ...

- **Any Java Class instance**

- **Sites that create sites:** MakeSemaphore, MakeChannel ...

...
Overview of Orc

- Orc program has
  - a set of definitions,
  - a goal expression, over sites and combinators.

- The goal expression is executed. Its execution
  - calls sites,
  - publishes values.
Structure of Orc Expression

- **Simple**: just a site call, \( CNN(d) \)
  Publishes the value returned by the site.

- **Composition** of two Orc expressions:
  
  \[
  \begin{align*}
  \text{do } f \text{ and } g \text{ in parallel} & \quad f \mid g \\
  \text{for all } x \text{ from } f \text{ do } g & \quad f > x > g \\
  \text{for some } x \text{ from } g \text{ do } f & \quad f < x < g
  \end{align*}
  \]

  Symmetric composition
  Sequential composition
  Pruning
Structure of Orc Expression

- **Simple**: just a site call, \( CNN(d) \)
  Publishes the value returned by the site.

- **Composition** of two Orc expressions:

  do \( f \) and \( g \) in parallel \( f | g \)  
  for all \( x \) from \( f \) do \( g \) \( f \gg x \gg g \)  
  for some \( x \) from \( g \) do \( f \) \( f < x < g \)  

Symmetric composition  
Sequential composition  
Pruning
Structure of Orc Expression

- **Simple**: just a site call, $CNN(d)$
  Publishes the value returned by the site.

- **Composition** of two Orc expressions:
  
  do $f$ and $g$ in parallel \hspace{1cm} f | g \hspace{1cm} \text{Symmetric composition}
  
  for all $x$ from $f$ do $g$ \hspace{1cm} f \gg x \gg g \hspace{1cm} \text{Sequential composition}
  
  for some $x$ from $g$ do $f$ \hspace{1cm} f \ll x \ll g \hspace{1cm} \text{Pruning}
Structure of Orc Expression

- **Simple**: just a site call, $CNN(d)$
  Publishes the value returned by the site.

- **Composition** of two Orc expressions:
  
  \[
  \begin{align*}
  \text{do } f \text{ and } g \text{ in parallel} & \quad f \mid g \quad \text{Symmetric composition} \\
  \text{for all } x \text{ from } f \text{ do } g & \quad f >x> g \quad \text{Sequential composition} \\
  \text{for some } x \text{ from } g \text{ do } f & \quad f <x< g \quad \text{Pruning}
  \end{align*}
  \]
Examples

- **CNN(d) | BBC(d):**
  Calls **CNN** and **BBC** simultaneously.
  Publishes up to two values.

- **(CNN(d) | BBC(d)) >x> Email(address, x):**
  Publishes up to two values from **Email**.

- **Email(address, x) <x< (CNN(d) | BBC(d)):**
  Publishes at most one value from **Email**.
Some Fundamental Sites

- \textit{if} \((b)\): boolean \(b\),
  returns a \texttt{signal} if \(b\) is true; remains \texttt{silent} if \(b\) is false.

- \textit{Rtimer} \((t)\): integer \(t\), \(t \geq 0\), returns a signal \(t\) time units later.

- \textit{stop}: never responds. Same as \textit{if} \((\text{false})\).

- \textit{signal}: returns a signal immediately. Same as \textit{if} \((\text{true})\).

Publish a signal at every time unit.

\texttt{def metronome()} = \texttt{signal} \mid \texttt{(Rtimer(1) \gg metronome())}
Laws Based on Kleene Algebra

(Zero and $|$)

(Commutativity of $|$)

(Associativity of $|$)

(Idempotence of $|$)

(Associativity of $≫$)

(Left zero of $≫$)

(Right zero of $≫$)

(Left unit of $≫$)

(Right unit of $≫$)

(Left Distributivity of $≫$ over $|$)

(Right Distributivity of $≫$ over $|$)

\[
\begin{align*}
&f \mid stop = f \\
&f \mid g = g \mid f \\
&(f \mid g) \mid h = f \mid (g \mid h) \\
&f \mid f = f \\
&(f \mid g) \mid h = f \mid (g \mid h) \\
&stop \mid f = stop \\
&f \mid stop = stop \\
&Signal \mid f = f \\
&f \mid x \mid \text{let}(x) = f \\
&f \mid (g \mid h) = (f \mid g) \mid (f \mid h) \\
&(f \mid g) \mid h = (f \mid h \mid g \mid h)
\end{align*}
\]
Additional Laws

(Distributivity over $\gg$) if $g$ is $x$-free
$$(f \gg g) <x< h = (f <x< h) \gg g$$

(Distributivity over $|$) if $g$ is $x$-free
$$(f \mid g) <x< h = (f <x< h) \mid g$$

(Distributivity over $<<$) if $g$ is $y$-free
$$(f <x< g) <y< h
= (f <y< h) <x< g$$

(Elimination of where) if $f$ is $x$-free, for site $M$
$$(f <x< M) = f \mid (M \gg stop)$$
Operational semantics of Orc

- **Traces** of an expression.
  Special treatment for unbound variables.

- Ordering of expressions based on subset ordering over traces.
  **Congruence**.

- Combinators are **monotonic** and **continuous**.

- Holds in the timed model too.
Encoding

To compute $1 + 2 + 3$:

$$add(1, x) < x < add(2, 3)$$
Functional Core Language

- **Data Types**: Number, Boolean, String, with usual operators
- **Conditional Expression**: `if E then F else G`
- **Data structures**: Tuple and List
- **Pattern Matching**
- **Function Definition; Closure**
Variable Binding; Silent expression

val \( x = 1 + 2 \)
val \( y = x + x \)
val \( z = x/0 \) \(-- \text{expression is silent}\)
val \( u = \text{if } (0 < 5) \text{ then } 0 \text{ else } z \)
Example: Fibonacci numbers

```python
def H(0) = (1, 1)
def H(n) =
    val (x, y) = H(n - 1)
    (y, x + y)

def Fib(n) =
    val (x, -) = H(n)
    x

-- Goal expression
Fib(5)
```
Translating Functional Core to Pure Orc

- Operators to Site calls:
  \[ 1 + (2 + 3) \] to \( \text{add}(1, x) < x < \text{add}(2, 3) \)

- if \( E \) then \( F \) else \( G \):
  \( (\text{if}(b) \gg F \mid \text{not}(b) > c > \text{if}(c) \gg G) < b < E) \)

- \text{val} \( x = G \) followed by \( F \):
  \( F < x < G \)

- Data Structures, Patterns: Site calls and variable bindings

- Function Definitions: Orc definitions
Comingling with Orc expressions

Components of Orc expression could be functional. Components of functional expression could be Orc.

\[
1 + 2 \mid 2 + 3,
\]

is \((\langle \langle \text{let}(x) \mid \text{let}(y) \rangle \rangle \langle x \rangle < \langle \text{add}(1, 2) \rangle \rangle \langle y \rangle < \langle \text{add}(2, 3) \rangle \rangle\).

Convention: whenever expression \( F \) appears in a context where a single value is expected, convert it to \( x \langle x \rangle < F \).

\[
(1 \mid 2) + (2 \mid 3),
\]

is \((\langle \text{add}(x, y) \rangle \langle x \rangle < \langle 1 \mid 2 \rangle \rangle \langle y \rangle < \langle 2 \mid 3 \rangle \rangle\).
Example: Fibonacci numbers

\[
\begin{align*}
def \, H(0) &= (1, 1) \\
def \, H(n) &= H(n - 1) \succ (x, y) \succ (y, x + y)
\end{align*}
\]

\[
\begin{align*}
def \, Fib(n) &= H(n) \succ (x, -) \succ x
\end{align*}
\]

– Goal expression
\[Fib(5)\]
def throw() = random(6) + 1

def exp(0) = 0
def exp(n) =
    (if throw() + throw() = 7 then 1 else 0)
    + exp(n - 1)

Dice Throw
def throw() = add(x, 1) < x < random(6)

def exp(n) =
    ( if(b) ≫ let(0)
        | not(b) > nb > if(nb)
        ( add(x, y)
            < x < ((if(bb) ≫ 1 | not(bb) > nbb) > if(nbb) ≫ 0)
            < bb < equals(p, 7)
            < p < add(q, r)
            < q < throw()
            < r < throw()
            < y < (exp(m) < m < sub(n, 1))
            )
    )  < b < equals(n, 0)
Fork-join parallelism

Call sites $M$ and $N$ concurrently.
Return their values as a tuple after both respond.

$((u, v) < u < M()) < v < N(),$ or simply

$(M(), N())$
Mutable Structures

val \( r = \text{Ref}() \)

\( r.\text{write}(3) \), or \( r := 3 \)
\( r.\text{read()} \), or \( r? \)

\[\text{def } \text{swapRefs}(x, y) = x? >z> x := y? \gg y := z\]
Random Permutation

\[ val\ N = 20 \quad -- \quad \text{size of permutation array} \]
\[ val\ ar = \text{fillArray}(\text{Array}(N), \lambda i . i) \]

\[ -- \quad \text{Randomize array } a \text{ of size } n, \quad n \geq 1 \]
\[ def\ randomize(1) = \text{signal} \]
\[ def\ randomize(n) = \]
\[ \quad \text{random}(n) \succ k \succ \]
\[ \quad \text{swapRefs}(ar(n - 1), ar(k)) \gg randomize(n - 1) \]

\[ \text{randomize}(N) \]
def search(key) = -- return true or false
searchstart(key) >(-, -, q) > (q ≠ null)

def insert(key) = -- true if value was inserted, false if it was there
searchstart(key) > (p, d, q) >
if q = null
then Ref() > r >
   r := (key, null, null) => update(p, d, r) => true
else false

def delete(key) =
**Semaphore**

\[
\text{val } s = \text{Semaphore}(2) \quad \text{-- } s \text{ is a semaphore with initial value 2}
\]

\[
\begin{align*}
& s.\text{acquire}() \\
& s.\text{release}()
\end{align*}
\]

Rendezvous:

\[
\begin{align*}
& \text{val } s = \text{Semaphore}(0) \\
& \text{val } t = \text{Semaphore}(0)
\end{align*}
\]

\[
\begin{align*}
& \text{def } \text{send} = t.\text{release}() \gg s.\text{acquire}() \\
& \text{def } \text{receive} = t.\text{acquire}() \gg s.\text{release}()
\end{align*}
\]

\(n\)-party Rendezvous using \(2(n - 1)\) semaphores.
val req = Buffer()
val cb = Counter()

def rw() =
    req.get() >(b, s)>
    ( if(b) => cb.inc() => s.release() => rw()
     | if(¬b) => cb.onZero()
        => cb.inc() => s.release() => cb.onZero() => rw()
    )

def start(b) =
    val s = Semaphore(0)
    req.put((b, s)) => s.acquire()

def quit() = cb.dec()
Scan, swap over array \( a \)

\[
def \ lr(i) = \begin{cases} 
  & \text{if } (i < t \land a(i) \leq p) \text{ then } lr(i + 1) \text{ else } i \\

def \ rl(i) = \begin{cases} 
  & \text{if } (a(i) > p) \text{ then } rl(i - 1) \text{ else } i \\

def \ swap(i, j) = a(i) > z > a(i) := a(j) \implies a(j) := z
\end{cases}
\]
Partition

\[ \text{def } \text{part}(p, s, t) = \]
\[ \text{def } \text{lr}(i) = \begin{cases} 
\text{lr}(i + 1) & \text{if } (i < t \land a(i) \leq p) \\
\text{i} & \text{else}
\end{cases} \]
\[ \text{def } \text{rl}(i) = \begin{cases} 
\text{rl}(i - 1) & \text{if } (a(i) > p) \\
\text{i} & \text{else}
\end{cases} \]

\((\text{lr}(s), \text{rl}(t)) > (s', t') \rangle \)

\(( \text{if } (s' + 1 < t') \Rightarrow \text{swap}(s', t') \Rightarrow \text{part}(p, s' + 1, t' - 1) \rangle \)

\| \text{if } (s' + 1 = t') \Rightarrow \text{swap}(s', t') \Rightarrow s' \rangle \]

\| \text{if } (s' + 1 > t') \Rightarrow t' \rangle \]

\)

Returns \( m \) where

\[ a(s) \cdots a(m) \leq p, \]
\[ a(m + 1) \cdots a(t) > p \]
def sort(s, t) =

if s ≥ t then signal

else part(a(s)?, s + 1, t) >m>

swap(m, s) »

(sort(s, m − 1), sort(m + 1, t)) »

signal

sort(0, a.length() − 1)
Putting the Pieces together

```python
def quicksort(a):
    def swap(x, y) = a(x)? >z> a(x) := a(y)? >> a(y) := z
    def part(p, s, t) =
        def lr(i) = if (i < t ∨ a(i)? ≤ p) then lr(i + 1) else i
        def rl(i) = if (a(i)? > p) then rl(i - 1) else i
        (lr(s), rl(t)) > (s', t') >
        (if (s' + 1 < t') >> swap(s', t') >> part(p, s' + 1, t' - 1)
         | if (s' + 1 = t') >> swap(s', t') >> s'
         | if (s' + 1 > t') >> t')
    def sort(s, t) =
        if s ≥ t then signal
        else part(a(s)?, s + 1, t) > m >
            swap(m, s) >>
            (sort(s, m - 1), sort(m + 1, t)) >>
            signal
        sort(0, a.length() - 1)
```
Remarks and Proof outline

- Concurrency without locks
- \( sort(m, n) \) sorts the segment; does not touch items outside the segment.
- Then, \( sort(s, m - 1) \) and \( sort(m + 1, t) \) are non-interfering.
- \( part(p, s, t) \) does not modify any value outside this segment. May read values.
Tony was/is a major inspiration in the development of Orc.