Quicksort: Combining Concurrency, Recursion, and Mutable Data Structures*

David Kitchin, Adrian Quark and Jayadev Misra

Abstract Quicksort [5] remains one of the most studied algorithms in computer science. It is important not only as a practical sorting method, but also as a splendid teaching aid for introducing recursion and systematic algorithm development. The algorithm has been studied extensively; so, it is natural to assume that everything that needs to be said about it has already been said. Yet, in attempting to code it using a recent programming language of our design, we discovered that its structure is more clearly expressed as a concurrent program that manipulates a shared mutable store, without any locking or explicit synchronization. In this paper, we describe the essential aspects of our programming language Orc [8], show a number of examples that combine its features in various forms, and then develop a concise description of Quicksort. We hope to highlight the importance of including concurrency, recursion and mutability within a single theory.

1 Introduction

Quicksort [5] remains one of the most studied algorithms in computer science. Its performance has been studied extensively, by Knuth [11] and Sedgewick [17] in particular. A variety of implementations exist on different architectures, and many variants of quicksort have been developed that improve its performance for specific platforms.

David Kitchin
University of Texas at Austin, e-mail: dkitchin@cs.utexas.edu

Adrian Quark
University of Texas at Austin, e-mail: quark@cs.utexas.edu

Jayadev Misra
University of Texas at Austin, e-mail: misra@cs.utexas.edu

* This work is partially supported by National Science Foundation grant CCF-0811536.

The structure of the algorithm has also been studied extensively, mainly in explaining program development and in teaching recursion. Yet, a concise description of the algorithm that does justice to its various aspects —mutable store, recursion and concurrency— has never been shown before. Functional programs typically do not admit in-situ permutation of data elements, imperative programs are typically sequential and do not highlight concurrency, and typical concurrency constructs do not combine well with recursion.

We have recently designed a process calculus [14] and a programming language based on it, called Orc [8]. We believe that the Orc coding of Quicksort, in Section 5, is both concise and faithful to the original intent of the algorithm.

This paper first presents Orc, starting with the Orc calculus, and then the programming language designed around the calculus. The calculus starts with the premise that concurrency is fundamental; sequential programming is a special case. The calculus itself is extremely small, consisting of four combinators (only three of which are essential for this paper) and a definition mechanism. It contains no data structuring, nor any notion of process, thread or communication. Yet, most aspects of concurrency, including synchronization, communication and interrupt, can be succinctly expressed within this calculus.

The calculus is next enhanced with a small functional language to ease writing of practical programs. The language includes basic operators, conditionals, some primitive data types, and pattern matching mechanisms. The enhancements are mere syntactic sugar; they can all be translated to the core Orc calculus, and that is how they are actually implemented. The programming model draws its power from external services, called *sites*, which may encode functionalities that are better expressed in other programming paradigms. The combinators allow these sites to be integrated into a full concurrent program.

The paper is structured as follows. In Section 2, we review the Orc concurrency calculus. Section 3 shows its expansion into a functional concurrent programming language, with a library of sites supporting time, mutable state, communication, and synchronization. Section 4 present a series of example programs using concurrency, recursion, and the additional capabilities provided by the site library. In Section 5, we present the Quicksort algorithm in Orc. Section 6 includes brief conclusion remarks.

For a more thorough review of the Orc language, see [8], from which Sections 2, 3, and 4 borrow substantially. We also encourage the reader to visit our website [16]; it hosts a comprehensive user guide [9], a community wiki, and a web-based interface for experimenting with Orc.

2 The Orc Concurrency Calculus

The Orc calculus is based on the execution of *expressions*. Expressions are built up recursively using Orc's concurrency *combinators*. When executed, an Orc expression invokes services and may *publish* values. Different executions of the same ex-

pression may have completely different behaviors; they may call different services, receive different responses from the same service, and publish different values. An expression is *silent* if it never publishes a value.

In order to invoke services, Orc expressions call *sites*. A site may be implemented on the client's machine or a remote machine. A site may provide any service; it could run sequential code, transform data, communicate with a web service, or be a proxy for interaction with a human user.

We describe three of the four concurrency combinators of Orc in this paper. Notable omissions in this paper are treatments of logical time (using site `Ltimer`) and halting (using the fourth concurrency combinator `;`). The operational and denotational semantics of the calculus appear in [7].

2.1 Site Calls

The simplest Orc expression is a *site call* $M(\overline{p})$, where M is a site name and $\overline{p}$ is a list of parameters, which are values or variables. The execution of a site call invokes the service associated with M, sending it the parameters $\overline{p}$. If the site responds, the call publishes that response. A site responds with at most one value.

Here are some examples of site calls.

`add(3,4)`	Add the numbers 3 and 4.
`CNN(d)`	Get the CNN news headlines for date `d`.
`Prompt("Name:")`	Prompt the user to enter a name on the console.
`random(10)`	Generate a random integer in the range 0..9.
`email(a,m)`	Send message `m` to email address `a`

Fundamental Sites

Though the Orc calculus itself contains no sites, there are a few fundamental sites that are so essential to writing useful programs that we always assume they are available. The site `let` is the identity site; when passed one argument, it publishes that argument, and when passed multiple arguments it publishes them as a tuple. The site `if` responds with a signal (a value which carries no information) if its argument is `true`, and otherwise it does not respond. The site call `Rtimer(t)` responds with a signal after exactly `t` time units.

`signal` and **`stop`**

For convenience, we allow two additional expressions: **`signal`** and **`stop`**. The expression **`signal`** just publishes a signal when executed; it is equivalent to `if(true)`. The expression **`stop`** is simply silent; it is equivalent to `if(false)`.

2.2 Combinators

Orc has four combinators to compose expressions: the parallel combinator `|`, the sequential combinator `>x>`, the pruning combinator[2] `<x<`, and the otherwise combinator `;`. We discuss only the first three in this paper; see [8] for more information on the otherwise combinator.

When composing expressions, the `>x>` combinator has the highest precedence, followed by `|`, then `<x<`.

Parallel Combinator

In F `|` G, expressions F and G execute independently. The sites called by F and G are the ones called by F `|` G and any value published by either F or G is published by F `|` G. There is no direct communication or interaction between F and G.

For example, evaluation of `CNN(d) | BBC(d)` initiates two independent computations; up to two values will be published depending on the number of responses received.

The parallel combinator is commutative and associative.

Sequential Combinator

In F `>x>` G, expression F is first evaluated. Each value published by F initiates a separate execution of G wherein `x` is bound to that published value. Execution of F continues in parallel with these executions of G. If F publishes no value, no execution of G occurs. The values published by the executions of G are the values published by F `>x>` G. The values published by F are consumed.

As an example, the following expression calls sites `CNN` and `BBC` in parallel to get the news for date `d`. Responses from either of these calls are bound to `x` and then site `email` is called to send the information to address `a`. Thus, `email` may be called 0, 1 or 2 times, depending on the number of responses received.

```
( CNN(d) | BBC(d) ) >x> email(a, x)
```

The sequential combinator is right associative, i.e. F `>x>` G `>y>` H is F `>x>` `(`G `>y>` H`)`. When `x` is not used in G, one may use the short-hand F `>>` G for F `>x>` G.

The sequential combinator generalizes the sequential composition of the traditional imperative languages for a concurrent world: if F publishes a single value and does nothing further, then F `>>` G behaves like the sequential program F;G.

Pruning Combinator

In F `<x<` G, both F and G execute in parallel. Execution of parts of F that do not depend on `x` can proceed, but site calls in F for which `x` is a parameter are suspended

[2] In previous publications, F `<x<` G was written as F **where** $x :\in G$.

until `x` has a value. If G publishes a value, then `x` is assigned that value, G's execution is terminated and the suspended parts of F can proceed. This is the only mechanism in Orc to block or terminate parts of a computation.

In contrast to sequential composition, the following expression calls `email` at most once.

```
email(a, x) <x< ( CNN(d) | BBC(d) )
```

The pruning combinator is left associative, i.e. F `<x<` G `<y<` H is `(`F `<x<` G`)` `<y<` H. When `x` is not used in F, one may use the short-hand F `<<` G for F `<x<` G.

The pruning combinator introduces eager concurrent evaluation. Later, we will see that expressions in the Orc language are often converted to pure Orc calculus using the pruning combinator; this introduces concurrency, even in the evaluation of arithmetic expressions, without programmer intervention.

2.3 Algebraic Properties of the Combinators

An operational semantics of Orc based on a labeled transition system appears in [18]. Employing bisimulation, we have proven the following algebraic properties of the combinators, some of which resemble laws of Kleene algebra (see [19] for these proofs). Below, we write "f is x-free" to mean that x does not occur as a free variable in f.

(Unit of $\mid$)	$f \mid \mathbf{stop} = f$
(Commutativity of $\mid$)	$f \mid g = g \mid f$
(Associativity of $\mid$)	$(f \mid g) \mid h = f \mid (g \mid h)$
(Left zero of $\gg$)	$\mathbf{stop} \gg x \gg f = \mathbf{stop}$
(Left unit of $\gg$)	$\mathbf{signal} \gg f = f$
(Right unit of $\gg$)	$f \gg x \gg let(x) = f$
(Associativity of $\gg$)	$(f \gg x \gg g) \gg y \gg h = f \gg x \gg (g \gg y \gg h)$, if h is x-free
(Distributivity of $\mid$ over $\gg$)	$(f \mid g) \gg x \gg h = (f \gg x \gg h \mid g \gg x \gg h)$
(Right unit of $\ll$)	$f \ll \mathbf{stop} = f$
(Associativity of $\ll$)	$(f \ll x \ll g) \ll y \ll h = f \ll x \ll (g \ll y \ll h)$, if f is y-free
(Commutativity of $\mid$ with $\ll$)	$(f \mid g) \ll x \ll h = (f \ll x \ll h) \mid g$, if g is x-free
(Commutativity of $\gg$ with $\ll$)	$(f \gg y \gg g) \ll x \ll h = (f \ll x \ll h) \gg y \gg g$, if g is x-free
(Commutativity of $\ll$ with $\ll$)	$((f \ll x \ll g) \ll y \ll h) = ((f \ll y \ll h) \ll x \ll g)$, if g is y-free and h is x-free

We can prove, for example, that $(f \ll x \ll g) = f \mid (\mathbf{stop} \ll x \ll g)$, if f is x-free. This follows from unit of $\mid$, commutativity of $\mid$, and commutativity of $\mid$ over $\ll$.

2.4 Definitions

An Orc expression may be preceded by a sequence of definitions of the form:

def $E(\overline{x})$ = F

This defines a function named E whose formal parameter list is $\overline{x}$ and body is expression F. Definitions may be recursive.

A call $E(\overline{p})$ executes the body F with the actual parameters $\overline{p}$ substituted for the formal parameters $\overline{x}$. A function call may publish more than one value; it publishes every value published by the execution of F. If multiple concurrent calls are made to a function E, all instances of the body F execute concurrently.

Unlike a site call, a function call does not require all of its arguments to have values. Suppose E is called when an actual parameter q, corresponding to a formal parameter y, does not have a value. As in the pruning combinator, the executions of parts of F that do not depend on y may proceed, and the parts that depend on y will block until q has a value, which is then substituted for y.

3 A Functional Concurrent Language

In the preceding section, we introduced a small concurrency calculus, which serves well as a formal model, but is not a practical language for writing larger programs. Now we describe a language by introducing constructs familiar from functional programming. We show how each construct can be represented in the Orc calculus, so that every program can be translated directly into an equivalent expression in the calculus that uses a small set of primitive sites for arithmetic or data structuring operations. We conclude with an example program and its translation into the calculus. For the details of the full language, see the Orc User Guide [9].

3.1 Functional Aspects of the Language

We introduce a series of expression forms, familiar from functional programming, each of which can be translated into an equivalent expression in the Orc calculus.

Values and Operators

The Orc language has three types of constants: numbers (5, -1, 2.71828, $\cdots$), strings (`"orc"`, `"ceci n'est pas une |"`, $\cdots$), and booleans (`true` and `false`). It provides typical arithmetic (`+ - * / ` $\cdots$), logical (`&& ||` $\cdots$), and comparison (`= < >` $\cdots$) operators. They are written infix with Java-like operator precedence. Parentheses can be used to override this precedence.

`(98+2)*17`	evaluates to `1700`.
`4 = 20 / 5`	evaluates to `true`.
`"leap" + "frog"`	evaluates to `"leapfrog"`.

The arithmetic, logical, and comparison operators translate directly to site calls; for example, `2+3` translates to `add(2,3)`, where `add` is simply a site that performs addition. A value `v` which occurs as an expression on its own becomes a site call `let(v)`.

Nested Expressions

Expressions may be nested, e.g. `2+(3+4)`. However, `2+(3+4)` cannot be translated directly to `add(2, add(3,4))`, since the calculus does not allow the call `add(3,4)` to appear as an argument. Instead, we use the pruning combinator, translating `2+(3+4)` to `add(2,z) <z< add(3,4)`, where `z` is a fresh variable name.

Expressions may even co-mingle concurrency combinators and functional constructs, as in `2 + (3 | 4)`. The same translation applies: `add(2,z) <z< (3 | 4)`. Since the pruning combinator `<x<` binds only the first value published by `3 | 4` to `x`, the expression could evaluate to either 5 or 6.

This translation can be applied to nested expressions at any depth, and it is the fundamental link between Orc as a concurrency calculus and Orc as a functional concurrent language.

Conditionals

A conditional expression is of the form **`if`** *E* **`then`** *F* **`else`** *G*. If *E* evaluates to `true`, then *F* is evaluated. If *E* evaluates to `false`, then *G* is evaluated. If *E* does not publish a value, neither *F* nor *G* is evaluated.

`if` `true` **`then`** `4` **`else`** `5`	evaluates to `4`.
`if` `0 < 5` **`then`** `0/5` **`else`** `5/0`	evaluates to `0`.
`if` `1 < 1/0` **`then`** `2` **`else`** `3`	is silent.

The conditional expression **`if`** *E* **`then`** *F* **`else`** *G* translates to:

`(` **`if`**`(b) >>` *F* `| not(b) >c>` **`if`**`(c) >>` *G* `) <b<` *E*

Recall that `if(true)` publishes a signal and `if(false)` is silent. The site `not` performs boolean negation.

Variables

We introduce and bind variables using a **`val`** declaration, as follows. Below, `x` and `y` are bound to `3` and `6`, respectively.

```
val x = 1 + 2
val y = x + x
```

Variables cannot be reassigned. If the same variable is bound again, subsequent references to that variable will use the new binding, but previous references remain unchanged. Variable bindings obey the rules of lexical scope.

The declaration **`val`** `x =` G, followed by expression F, translates to:

F `<x<` G

All the rules that apply to the pruning combinator apply to `val`, and it is permissible to write any Orc expression, even one that publishes multiple values, in a `val`. One of the most common Orc programming idioms is to write a `val` to choose the first available publication of a concurrent expression:

```
val url = Google("search term") | Yahoo("search term")
```

Data Structures

The Orc language supports two types of data structures: *tuples*, such as `(3, 7)` or `("tag", true, false)`, and finite *lists*, such as `[4,4,1]` or `["example"]` or `[]`. A tuple or list containing expressions to be evaluated is itself an expression; each of the expressions is evaluated, and the result is a tuple or list of those results.

`[1,2+3]`	evaluates to `[1,5]`.
`(3+4, if true then "yes" else "no")`	evaluates to `(7, "yes")`.

Tuples and lists can contain any value, including other tuples or lists.

The prepend (*cons*) operation on lists is written `x:xs`, where `xs` is a list and `x` is some element to be prepended to that list.

`[3,5] >t> 1:t`	evaluates to `[1,3,5]`.

Data structures are created by site calls. The site `let` creates tuples directly. The site `nil` returns the empty list when called. The site `cons` implements the cons operator and is also used to construct list expressions. For example, `[1,2]` translates to `cons(1,s) <s< cons(2,t) <t< nil()`.

Patterns

We can bind parts of data structures to variables using *patterns*. We write _ for the wildcard pattern.

Patterns may replace variables in the `>x>` and `<x<` combinators. If a publication does not match the pattern of a `>x>` combinator, the publication is ignored, and no new instance of the right hand expression is executed. For the `<x<` combinator, the publication is ignored, and the right hand expression continues to run.

`(3,4) >(x,y)> x+y`	publishes `7`.
`x <(0,x)< ((1,0) | (0,1))`	publishes `1`.

Since the `val` declaration is simply a different form of the `<x<` combinator, patterns may replace variables in `val` as well:

`val` `(x,y) = (2+3,2*3)`
binds `x` to `5` and `y` to `6`.

`val` `[a,_,c] = "one":["two", "three"]`
binds `a` to `"one"` and `c` to `"three"`.

`val` `((a,b),c) = ((1, true), (2, false))`
binds `a` to `1`, `b` to `true`, and `c` to `(2,false)`.

Patterns can be translated into a set of calls to pattern deconstruction sites followed by a set of variable bindings to match up each of the pieces with the appropriate variable names.

Functions

Functions are defined using the keyword **`def`**, and are identical to definitions in the Orc calculus. Definitions may be recursive, and groups of definitions may be mutually recursive.

```
def sumto(n) = if n <= 0 then 0 else n + sumto(n-1)
```

Functions can be defined as a series of *clauses*, each of which has a different list of patterns for its formal parameters. When such a function is called, the function body used for the call is that of the first clause whose formal parameter patterns match the actual parameters.

```
def fib(0) = 0
def fib(1) = 1
def fib(n) = if (n < 0) then 0 else fib(n-1) + fib(n-2)
```

The function `fib` may also be written more efficiently, as follows:

```
def fibpair(0) = (0,1)
def fibpair(n) = fibpair(n-1) >(a,b)> (b,a+b)
def fib(n) = if (n < 0) then 0 else fibpair(n) >(x,_)> x
```

Defining a function creates a value called a *lexical closure*; the name of the function is a variable and its bound value is the closure, which records all of the current bindings for free variables in the function body.

Since a closure is a value, it can be passed as an argument to another function, thus allowing us to define *higher-order* functions. As an example, here is the classic `map` function; see additional examples in Sections 4.1.3 and 4.3.3.

```
def map(f,[]) = []
def map(f,x:xs) = f(x):map(f,xs)
```

Note the important distinction between `f` and `f(x)`; the former is a variable whose bound value is a function (closure), and the latter is a call to that function.

3.2 Implicit Concurrency: An Example

We show an Orc program that does not use any of the concurrency combinators explicitly. In fact, the program is entirely functional, with the sole exception of the site call `random(6)`, which returns a random integer between `0` and `5`. Yet, each nested expression translates into a use of the pruning combinator, making this program implicitly concurrent without any programmer intervention.

The program runs a series of experiments. Each experiment consists of rolling a pair of dice. An experiment succeeds if the total shown by the two dice is `c`. The function `exp(n,c)` returns the number of successes in `n` experiments.

```
-- return a random number between 1 and 6
def toss() = random(6) + 1

def exp(0,_) = 0
def exp(n,c) = exp(n-1,c) + (if toss()+toss() = c then 1 else 0)
```

In `exp(n,c)`, the two expressions `exp(n-1,c)` and `if toss()+toss() = ...` may be executed concurrently, as may both calls to `toss`. Therefore, all $2n$ calls to `toss` may be executed concurrently. This is clearly seen in the translation, given below, of this program into the Orc calculus. Here, site `add` returns the sum of its arguments, `sub(x,y)` returns `x-y`, `not(b)` returns the negation of `b`, and `equals` returns `true` iff its two arguments are equal.

```
def toss() = add(x,1) <x< random(6)

def exp(n,c) =
   ( if(b) >> let(0)
   | not(b) >nb> if(nb) >>
     ( add(x,y)
         <x< ( exp(m,c) <m< sub(n,1) ) )
         <y< ( ( if(bb) >> 1 | not(bb) >nbb> if(nbb) >> 0 )
                   <bb< equals(p,c)
                     <p< add(q,r)
                       <q< toss()
                       <r< toss() )

   ) <b< equals(n,0)
```

3.3 Site library

We have implemented a library of useful sites. We introduce a few essential sites here, and we also note a few properties of sites that were not previously discussed.

Sites are first-class values

In both the Orc calculus and the Orc programming language, sites are first-class values; they may be bound to variables, passed as arguments, published, and returned by site calls. It is very important that sites can be published by other sites, as this allows the use of "factory" sites, which create new sites such as mutable references or communication channels.

Sites may have methods

Sites may represent objects with multiple methods, in an object-oriented style. We access methods on sites using a special form of site call, as in `c.put(4)`, which accesses the `put` method of channel `c` and calls it as a site, with argument `4`.

This call form, like every other new syntactic form introduced so far, can be encoded in the Orc calculus. The site `c` is sent a special value called a *message*, in this case the `"put"` message. The site responds to that message with another site which will execute the desired method when called. So `c.put(4)` translates to `c("put") >x> x(4)`.

Time

Orc is designed to communicate with the external world, and one of the most important characteristics of the external world is the passage of time. Orc implicitly accounts for the passage of time by interacting with external services that may take time to respond. However, Orc can also explicitly wait for a specific amount of time, using the special site `Rtimer`. The call `Rtimer(`t`)`, where t is an integer, responds with a signal exactly t milliseconds later[3].

We can use `Rtimer` together with the `<x<` combinator to enforce a timeout. Continuing with the example from Section 2.2, we can query `BBC` for a headline, but allow a default response if `BBC` does not respond within 5 seconds.

```
email(a, x) <x< (BBC(d) | Rtimer(5000) >> "BBC timed out.")
```

References

Orc does not have mutable variables. Mutable state is provided by sites instead. The `Ref` site is used to create new mutable references, which are used in a style similar to Standard ML's `ref` [15].

A call to `Ref` may include an argument specifying the initial contents of the reference; if none is given, then the reference's value is undefined. Given a reference `r`, `r.write(v)` overwrites the current value stored in `r`, changing it to `v`, and returns a signal; `r.read()` publishes the current value stored in `r`. If `r` is undefined, `r.read()` blocks until a value is written into `r`.

We write `r := v` as syntactic sugar for `r.write(v)`, and `r?` for `r.read()`.

[3] An implementation can only approximate this guarantee.

Arrays

The `Array` site creates new mutable arrays. Calling `Array(n)`, where `n` is the size of the array to be created, returns an array `a` with indices `0` through `n-1`, where the element values are undefined. Elements of array `a` are accessed by a site call, `a(i)`, which returns a reference to the *i*th element. That reference can then be read or written just like any reference created by `Ref`. The expression `a.length()` returns the length of the array.

`Array(3) >a> a(0):= true >> a(1):= false >> a(1)?`
publishes `false`.

`Array(3) >a> a(a.length()-1)?`
blocks until `a(2)` has a value.

Semaphores

Unlike other concurrent languages, Orc does not have any built-in locking mechanisms. Instead, it uses the `Semaphore` site to create semaphores which enable synchronization and mutual exclusion. `Semaphore(k)` creates a semaphore with the initial value `k` (i.e. it may be acquired by up to `k` parties simultaneously). Given a semaphore `s`, `s.acquire()` attempts to acquire `s`, reducing its value by one if it is positive, or blocking if its value is zero. The call `s.release()` releases `s`, increasing its value by one. The implementation of `Semaphore` guarantees strong fairness, i.e. if the semaphore value is infinitely often nonzero, then every call to `acquire` will eventually succeed.

We show below a function that returns an array of `n` semaphores, each with initial value 0.

```
def semArray(n) =
  val a = Array(n)
  def populate(0) = signal
  def populate(i) = a(i-1) := Semaphore(0) >> populate(i-1)
  populate(n) >> a
```

In practice, semaphores and other synchronization sites are only needed when resolving resource conflicts, i.e. concurrent calls to a site that has mutable state. Orc programs with implicit concurrency do not require these arbitration mechanisms.

Channels

Orc has no communication primitives like π-calculus channels [13] or Erlang mailboxes [1]. Instead, it makes use of sites to create channels of communication.

The most frequently used of these sites is `Buffer`, which publishes a new asynchronous FIFO channel. That channel is a site with two methods: `get` and `put`. The call `c.get()` takes the first value from channel `c` and publishes it, or blocks until a value becomes available. The call `c.put(v)` puts `v` as the last item of `c` and publishes a signal. A channel is a value, so it can be passed as an argument

4 Example Programs

In this section, we present a number of small programs, demonstrating how Orc combines concurrency and recursion, integrates real time, manipulates mutable state, and performs complex synchronizations.

4.1 Examples using Concurrency and Recursion

These examples implement some common idioms of concurrent and functional programming. Despite the austerity of Orc's combinators, we are able to encode a variety of idioms concisely.

4.1.1 Fork-Join

One of the most common concurrent idioms is a *fork-join*: evaluate two expressions F and G concurrently and wait for a result from both before proceeding. This is easily expressed in Orc:

`(`*F*`, `*G*`)`

Recall that this is equivalent to:

`( (x,y) <x< `*F*` ) <y< `*G*

This encoding takes advantage of the fact that a tuple is constructed by a site call, which must wait for all of its arguments to become available. In fact, any operator or site call may serve to join forked expressions. For example, if F and G each publish a number and we wish to output their maximum value, we simply write `max(F, G)`, where `max` returns the maximum of its arguments. We extend this example below.

Simple Parallel Auction

Orc programs often use fork-join together with recursion to dispatch many tasks in parallel and wait for all of them to complete. Suppose we have a list of bidders in a sealed-bid, single-round auction. Calling `b.ask()` requests a bid from the bidder `b`. We want to ask for one bid from each bidder and then publish the highest bid. The function `auction` performs this task:

```
def auction([]) = 0
def auction(b:bs) = max(b.ask(), auction(bs))
```

Note that all bidders are called simultaneously. Also note that if some bidder fails to return a bid, then the auction will never complete. Section 4.2.1 presents a different solution that addresses the issue of non-termination by using timeout.

4.1.2 Parallel Or

"Parallel or" is a classic idiom of parallel programming. The "parallel or" operation executes two expressions F and G in parallel, each of which may publish a single boolean, and returns the disjunction of their publications as soon as possible. If one of the expressions publishes `true`, then the disjunction is `true`, so it is not necessary to wait for the other expression to publish a value. This holds even if one of the expressions never publishes a value.

The "parallel or" of expressions F and G may be expressed in Orc as follows:

```
val a = F
val b = G
val result = (a || b) | if(a) >> true | if(b) >> true
result
```

The expression `(a || b)` waits for both `a` and `b` to become available and then publishes their disjunction. However, if either `a` or `b` is true we must publish `true` immediately, regardless of whether the other variable has a value. Therefore we run `if(a) >> true` and `if(b) >> true` in parallel. Since more than one of these expressions may publish `true`, we bind only the first result to `result`. The value of the whole expression is simply the value bound to `result`.

A Generalization of Parallel-or

We now generalize the parallel-or problem. As before, let expressions F and G return values for variables `x` and `y`. The values need not be booleans. It is required to compute `R(x,y)`, where `R` is a site. However, `x` and `y` might not both become available, so we would like to compute `M(x)` or `N(y)` as approximations of `R(x,y)`, as the values become available. We call `M(x)`, `N(y)` and `R(x,y)` as soon as their parameters have values, subject to the following constraints:

1. If `M(x)` has been called and it responds before y acquires a value, then `M(x)` is published as the only result (for `N(y)`, dually).
2. If both `x` and `y` have values before either `M(x)` or `N(y)` has responded, then `R(x,y)` is called and its value is published.
3. Both `M(x)` and `N(y)` should not be called, because then both `x` and `y` are available and `R(x,y)` could be called.

The following program almost solves the problem. Here boolean `b` is true if one of the sites, `M` or `N`, has responded before both `x` and `y` have values. In the latter case (`b` is false), `R(x,y)` is called and its response, if any, is published. In the former case, the response from `M` or `N` is published.

```
val (v,b) = (M(x),false) | (N(y),false) | ((x,y),true)
if b then R(x,y) else v
```

There is a possibility of a race condition. Suppose that `x` becomes available and then `M(x)` is called. Now, `y` becomes available, `N(y)` is called and a response is

received from it before `((x,y),true)` publishes a value. This would violate constraint (3). The possibility of an immediate response is remote for a remote site `N` (pun intended). Yet, we would like to eliminate this possibility because `N` could be a local site. The following solution adds an extra boolean variable c that records which of the parameters, `x` or `y`, first acquired a value, and correspondingly, only `M` or `N` is called. The rest of the solution is borrowed from the previous case.

```
val c = x >> true | y >> false
val (v,b) = if(c)  >> (M(x),false)
          | if(~c) >> (N(y),false)
          | ((x,y),true)
if b then R(x,y) else v
```

4.1.3 Fold

We consider various concurrent implementations of the classic "list fold" function, defined by `fold(f, [`x_1`, ... , `x_n`]) = f(`x_1`, f(`x_2`, ... f(`x_{n-1}`, `x_n`) ... )`. Here is a simple functional implementation:

```
def fold(_, [x])  = x
def fold(f, x:xs) = f(x, fold(xs))
```

This is a seedless fold (sometimes called `fold1`) which requires that the list be nonempty, and uses its first element as the seed. This implementation is short-circuiting — it may finish early if the reduction operator `f` does not use its second argument — but it is not concurrent; no two calls to `f` can proceed in parallel. However, if `f` is associative, we can overcome this restriction and implement fold concurrently. If `f` is also commutative, we can further increase concurrency.

Associative Fold

We define `afold(f,xs)` where `f` is an associative binary function and `xs` is a nonempty list. The implementation iteratively reduces `xs` to a single value. Each iteration applies the auxiliary function `step`, which reduces adjacent pairs of items to single values.

```
def afold(f, [x]) = x
def afold(f, xs) =
  def step([]) = []
  def step([x]) = [x]
  def step(x:y:xs) = f(x,y):step(xs)
  afold(f, step(xs))
```

Notice that `f(x,y):step(xs)` is an implicit fork-join, as described in Section 4.1.1. Thus, the call `f(x,y)` executes concurrently with the recursive call `step(xs)`. As a result, all calls to `f` execute concurrently within each iteration of `afold`.

Associative, Commutative Fold

We can make the implementation even more concurrent when the fold operator is both associative and commutative. We define `cfold(f,xs)`, where `f` is an associative and commutative binary function and `xs` is a non-empty list. The implementation initially copies all list items into a buffer in arbitrary order using the auxiliary function `xfer`, counting the total number of items copied. The auxiliary function `combine` repeatedly pulls pairs of items from the buffer, reduces them, and places the result back in the buffer. Each pair of items is reduced in parallel as they become available. The last item in the buffer is the result of the overall fold.

```
def cfold(f, xs) =
  val c = Buffer()

  def xfer([])   = 0
  def xfer(x:xs) = c.put(x) >> stop | xfer(xs)+1

  def combine(0) = stop
  def combine(1) = c.get()
  def combine(m) = c.get() >x> c.get() >y>
                   ( c.put(f(x,y)) >> stop | combine(m-1))

  xfer(xs) >n> combine(n)
```

4.2 Examples using Time

These examples demonstrate how Orc programs can integrate real time to detect time outs, and execute expressions at regular time intervals, using the site `Rtimer` described in Section 3.3.

4.2.1 Timeout

Timeout, the ability to execute an expression for at most a specified amount of time, is an essential ingredient of fault-tolerant and distributed programming. Orc accomplishes timeout using pruning and the `Rtimer` site, as we saw in Section 3.3; we further develop that technique in these examples.

Auction with Timeout

The auction example in Section 4.1.1 may never finish if one of the bidders does not respond. We add a timeout so that each bidder has at most 8 seconds to respond:

```
def auction([]) = 0
def auction(b:bs) = max(b.ask() | Rtimer(8000) >> 0, auction(bs))
```

This version of the auction is guaranteed to complete within 8 seconds.

Priority

We can use `Rimer` to give a window of priority to one computation over another. In this example, we run expressions F and G concurrently. For one second, F has priority; F's result is published immediately, but G's result is held until the time interval has elapsed. If neither F nor G publishes a result within one second, then the first result from either is published.

```
val x = F
val y = G
let(x | Rtimer(1000) >> y)
```

Detecting Timeout

Sometimes, rather than just yielding a default value, we would like to determine whether an expression has timed out, and if so, perform some other computation. To detect the timeout, we pair the result of the original expression with `true` and the result of the timer with `false`. Thus, if the expression does time out, then we can distinguish that case using the boolean value.

Here, we run expression F with a time limit t. If it publishes within the time limit, we bind its result to `r` and execute G. Otherwise, we execute H.

```
val (r, b) = (F, true) | (Rtimer(t), false)
if b then G else H
```

4.2.2 Metronome

A timer can be used to execute an expression repeatedly at regular intervals. We define a function `metronome(t)`, which publishes a signal every `t` time units.

```
def metronome(t) = signal | Rtimer(t) >> metronome(t)
```

The following example publishes "tick" once per second and "tock" once per second after an initial half-second delay. The publications alternate: "tick tock tick tock ...". Note that this code is not recursive; the recursion is entirely contained within `metronome`.

```
  metronome(1000) >> "tick"
| Rtimer(500) >> metronome(1000) >> "tock"
```

4.3 Examples using Mutable State

These examples show how Orc can manipulate mutable state, such as the reference cells and arrays described in Section 3.3. Recall that `x?` is syntactic sugar for `x.read()`, and `x := y` for `x.write(y)`. Also recall that the expression `a(i)` returns a reference to the element of array `a` at index `i`; array indices start from 0.

4.3.1 Simple Swap

The following function takes two references as arguments, exchanges their values, and returns a signal.

```
def swap(a, b) = (a?, b?) >(x,y)> (a := y, b := x) >> signal
```

4.3.2 Array Permutation

The following function randomly permutes the elements of an array in place. It uses the helper function `randomize`, which for each index `i` in the array, generates a random number `j` between `0` and `i` inclusive, and swaps `a(i)` with `a(j)`.

```
-- Randomize array a of size n, n >= 1
def permute(a) =
  def randomize(0) = signal
  def randomize(i) = random(i+1) >j>
                     swap(a(i),a(j)) >>
                     randomize(i-1)
  randomize(a.length() - 1)
```

Since `random` returns values from a uniform distribution, each possible permutation of the array is equally likely. This algorithm originally appears in [3, 10].

The technique we use here —traversing an array recursively and calling `swap` to exchange its elements— is crucial for our Quicksort implementation.

4.3.3 Applicative Map

The `map` function, shown in Section 3.1, applies a function to each element in a list and returns a new list populated with the results; it is a common idiom in pure functional programming. When manipulating mutable arrays, it is often helpful to perform a map operation in place: apply the function to each element of the array, overwriting the previous contents. The function `inplacemap(f,a)`, defined below, applies the function `f` to the array `a` in this way.

```
def inplacemap(f,a) =
  def mapstep(0) = signal
  def mapstep(i) = (a(i-1) >e> e := f(e?), mapstep(i-1))
  mapstep(a.length()) >> signal
```

The function `f` is applied to each element concurrently; when all calls to `f` complete, `inplacemap` returns a signal. Note that if an element `a(i)` of the array is undefined, `a(i)?` will block.

Suppose we define the functions `zero` and `incr` in the following way:

```
def zero(x) = 0
def incr(x) = x+1
```

Then `inplacemap(zero,a)` zeroes out the array `a`, and `inplacemap(incr,a)` increments the value of each element of `a` by 1.

4.4 Examples using Synchronization and Communication

Synchronization and communication are fundamental to concurrent computing. We implement some examples of synchronization –the rendezvous mechanism [6, 12] and a solution to the readers-writers problem [2]– and show how communicating processes [6] may be programmed in Orc.

4.4.1 Rendezvous

The concept of *rendezvous* between two parties was first introduced by Hoare and Milner as a form of process synchronization. Orc does not include rendezvous as a primitive concept, but we can program it using semaphores. First, we show a two party rendezvous, and then generalize it to $(n+1)$-party rendezvous, for $n \geq 1$.

A rendezvous occurs between a sender and a receiver when both of them are waiting to perform their respective operations. They each wait until they complete the rendezvous, and then they can proceed with their computations. A rendezvous involves synchronization and data transfer. In the solution below, first we show only the synchronization, and later data transfer. Potentially many senders and receivers may simultaneously wait to rendezvous, but each can rendezvous with exactly one other party.

Senders and receivers call the functions `send` and `receive`, respectively, when they are ready to rendezvous. The solution employs two semaphores, `up` and `down`, which are acquired and released in a symmetric manner. (The roles of sender and receiver are symmetric; so the two function bodies may be exchanged.) It can be shown that this solution synchronizes a pair of sender and receiver, each of them receives a signal following a synchronization, and that it leaves the semaphores in their original states (with value 0) following the synchronization. We expect each semaphore to be binary-valued, yet this is not a requirement. For general semaphores, there is still pairwise synchronization, though it cannot be ascertained which sender has synchronized with which receiver.

```
val up   = Semaphore(0)
val down = Semaphore(0)
def send() = up.release() >> down.acquire()
def recv() = up.acquire() >> down.release()
```

In order for the sender to send data value `v`, replace semaphore `up` by buffer `b`, and modify the programs:

```
def send(v)    = b.put(v) >> down.acquire()
def receive() = b.get() >x> down.release() >> x
```

The given solution can be generalized to the case where the senders and receivers belong to specific groups, and a rendezvous occurs only between members of a group. In that case, each group uses its own pair of semaphores and corresponding definitions.

$(n+1)$-party Rendezvous

We generalize the rendezvous algorithm given above to synchronize $n+1$ parties, $n \geq 1$, using $2n$ semaphores. We create two arrays of semaphores, `up` and `down`, using the `semArray` function defined in Section 3.3. The algorithm is reminiscent of 2-phase commit protocol in databases. Each of the $n+1$ parties calls a function when it is ready to synchronize, like the sender and the receiver above. One party is designated the coordinator, and it calls a special function `coord`; all others call `read`. Any other party with index `i`, where `0 ≤ i < n`, first releases semaphore `up(i)` and then waits to acquire `down(i)`. Function `coord` first acquires all the `up` semaphores and then releases all the `down` semaphores. The 2-party rendezvous is a special case where the receiver played the role of the coordinator.

```
val up = semArray(n)
val down = semArray(n)
def ready(i) = up(i).release >> down(i).acquire
def coord() =
  def Acq(0) = signal
  def Acq(k) = up(k-1).acquire >> Acq(k-1)
  def Rel(0) = signal
  def Rel(k) = (down(k-1).release,Rel(k-1)) >> signal
  Acq(n) >> Rel(n)
```

4.4.2 Readers-Writers Synchronization

We present a solution to the classical Readers-Writers synchronization problem [2]. Processes, called *readers* and *writers*, share a resource such that concurrent reading is permitted but a writer needs exclusive access. We present a starvation-free solution consisting of three functions: `start`, `end`, and `manager`. Readers and writers call `start`, a blocking operation, to request access; readers call `start(true)` and writers call `start(false)`. Function `start` publishes a signal when the resource can be granted. Readers and writers call `end()` to release the resource. Function `manager` runs concurrently with the rest of the program to grant the requests.

A call to `start(b)` adds a request to channel `q`. Function `manager` reads from `q`, decides when the request can be granted, and then calls back the requester. We employ semaphores for callback. Specifically,

```
def start(b) = Semaphore(0) >s> q.put((b,s)) >> s.acquire()
```

Function `manager` releases `s` when it can grant the request. Since `s` has initial value 0, `s.acquire()` blocks until the request is granted.

To count the number of active readers and writers, we employ a *counter* `c`, a mutable object on which three methods are defined: (1) `c.inc()` adds 1 to the counter value, (2) `c.dec()` subtracts 1 from the counter value, and (3) `c.onZero()` sends a signal only when the counter value is zero. The first two are non-blocking operations, and the last one is blocking. Though `onZero()` sends a signal only when the counter value is zero, the value may be non-zero by the time the recipient re-

ceives the signal. There is a weak fairness guarantee: if the counter value remains 0 continuously, a signal is sent to some caller of `onZero()`. The counter is initially 0.

The code for `end` merely decrements the counter:

```
def end() = c.dec()
```

The manager is an eternal loop structured as follows:

```
def manager() =
  q.get() >(b,s)> if b then read(s) else write(s) >> manager()
```

The invariants in each iteration of `manager` are that (1) there are no permitted writers, and (2) the counter value is the number of permitted readers. We use these invariants in the implementations of `read` and `write`.

A reader can always be granted permission to execute, from invariant (1). To satisfy invariant (2), the counter value must be incremented.

```
def read(s) = c.inc() >> s.release()
```

A writer can be granted permission only if there are no active readers. To satisfy invariant (1), the execution of `write` can terminate only when there are no active writers.

```
def write(s) =
  c.onZero() >> c.inc() >> s.release() >> c.onZero()
```

We start execution of an instance of `manager` by writing:

```
val _ = manager()
```

4.4.3 Process Network

Process networks [6] and actors [4, 1] are popular models for concurrent programming. Their popularity derives from the structure they impose on a concurrent computation. Different aspects of the computation are partitioned among different processes (actors), and the processes communicate through messages over channels. This simple programming model allows a programmer to focus attention on one aspect of a problem at a time, corresponding to a process. Additionally, interference and race conditions among processes, the bane of concurrent programming, are largely eliminated by restricting communication to occur through messages.

We show how this programming model can be incorporated within Orc. Channels are created by calls to `Buffer`. Processes are represented as Orc definitions, which share these channels. The entire process network is the parallel composition of these processes. Below, we restrict ourselves to FIFO channels though other communication protocols can be defined in Orc.

As an example, consider a transformer process `P` that reads inputs one at a time from channel `in`, transforms each input to a single output by calling function (or site) `transform`, writes the result on channel `out`, and repeats these steps forever.

```
def P(c,d) = c.get() >x> transform(x) >y> d.put(y) >> P(c,d)
P(in,out)
```

Next, we build a small network of such processes. The network has two processes, and both read from `in` and write to `out`.

```
P(in,out) | P(in,out)
```

Here, the two processes may operate at arbitrary speeds in removing items from `in` and writing to `out`. Therefore, the order of items in the input channel is not preserved with the corresponding outputs in `out`.

Next, consider adding a *balancer* process that reads from `in` and randomly assigns the input to one of the processes for transformation, as a form of load balancing. Again, the processes write their results to `out`. We define two internal channels `in'` and `in''` which link *balancer* to the transformer processes.

```
def balancer(c,c',c'') =
  c.get() >x>
  (if random(2) = 0 then c'.put(x) else c''.put(x)) >>
  balancer(c,c',c'')

val in' = Buffer()
val in'' = Buffer()
balancer(in,in',in'') | P(in',out) | P(in'',out)
```

As a variation, consider a network in which the two transformer processes operate as before, but the order of inputs is preserved in the output. We replace the *balancer* process with a *distributor* process that sends alternate input items along `in'` and `in''`. The transformer processes write their outputs to two internal channels `out'` and `out''`. And, we add a *collector* process that copies the values from `out'` and `out''` alternately to `out`.

```
def distributor(c,c',c'') =
  c.get() >x> c'.put(x) >>
  c.get() >y> c''.put(y) >>
  distributor(c,c',c'')

def collector(d',d'',d) =
  d'.get() >x> d.put(x) >>
  d''.get() >y> d.put(y) >>
  collector(d',d'',d)

val out' = Buffer()
val out'' = Buffer()

  distributor(in,in',in'')
| P(in',out') | P(in'',out'')
| collector(out',out'',out)
```

We have shown some very simple networks here; in particular, the networks are acyclic and a-priori bounded in size. See [16] for networks in which arbitrarily many processes are dynamically initiated, interrupted, resumed or terminated. The networks may be structured in a hierarchy where a process itself may be a network to any arbitrary depth, and connections among network components are established statically by naming explicit channels as shown, or by sending a channel name as a data item.

5 Quicksort in Orc

The original quicksort algorithm [5] was designed for efficient execution on a uniprocessor. Encoding it as a functional program typically ignores its efficient rearrangement of the elements of an array. Further, no known implementation highlights its concurrent aspects. The following program attempts to overcome these two limitations. The program is mostly functional in its structure, though it manipulates the array elements in place. Presenting quicksort as a concurrent program shows its structure more clearly, and allows a number of options for implementation on a multiprocessor.

We define a `quicksort` function, which takes as its only argument an array `a`, and sorts that array in place. When the sort is complete, it returns a signal.

```
def quicksort(a) = ...
```

Within the body of `quicksort`, we define an auxiliary function `part(p,s,t)` that partitions the subarray of `a` defined by indices `s` through `t` into two partitions, one containing values $\leq$ `p` and the other containing values $>$ `p`. One of the partitions may be empty. The call `part(p,s,t)` returns an index `m` such that `a(i)` $\leq$ `p` for all `s` $\leq$ `i` $\leq$ `m`, and `a(j)` $>$ `p` for all `m` $<$ `j` $\leq$ `t`.

```
def part(p, s, t) = ...
```

To create the partitions, `part` calls two auxiliary functions `lr` and `rl`. These functions scan the subarray from the left and right respectively, looking for the first out-of-place element. Function `lr` returns the index of the leftmost item that exceeds `p`, or simply `t` if there is none. Function `rl` returns the index of the rightmost item that is less than or equal to `p`, or simply `s-1` if there is none (the value at `a(s-1)` is assumed to be $\leq$ `p`).

```
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
```

Observe that `lr` and `rl` may safely be executed concurrently, since they do not modify the array elements.

Once two out-of-place elements have been found, they are swapped using the function `swap` defined in Section 4.3.1, and then the unscanned portion of the subarray is partitioned further. Partitioning is complete when the entire subarray has been scanned. Here is the body of the `part` function:

```
(lr(s), rl(t)) >(s', t')>
  ( if(s' + 1 < t') >> swap(a(s'), a(t')) >> part(p, s'+1, t'-1)
  | if(s' + 1 = t') >> swap(a(s'), a(t')) >> s'
  | if(s' + 1 > t') >> t'
  )
```

Observe that in the body of `part`, we use three parallel calls to the `if` site, with mutually exclusive conditions, each followed by `>>` and another expression. This is a representation of Dijkstra's *guarded commands* in Orc, using `if` to represent a guard, followed by `>>` and a consequent. Also observe that the second guarded

command can be eliminated by replacing the first guard by `s' + 1 <= t'`; this incurs a slight performance penalty.

The main sorting function `sort(s,t)` sorts the subarray given by indices `s` through `t` by calling `part` to partition the subarray and then recursively sorting the partitions concurrently. It publishes a signal on completion.

```
def sort(s, t) =
     if s >= t then signal
     else part(a(s)?, s+1, t) >m>
          swap(a(m), a(s)) >>
          (sort(s, m-1), sort(m+1, t)) >>
          signal
```

The body of `quicksort` is just a call to `sort`, selecting the whole array:

```
sort(0, a.length()-1)
```

Here is the `quicksort` program in its entirety.

```
def quicksort(a) =

  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(a(s'),a(t')) >> part(p,s'+1,t'-1)
    | if (s' + 1 = t') >> swap(a(s'),a(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(a(m), a(s)) >>
          (sort(s, m-1), sort(m+1, t)) >>
          signal

  sort(0, a.length()-1)
```

6 Why Orc?

Much like other process algebras, the Orc calculus was designed to study the appropriateness of certain combinators for concurrent computing. Unlike most other process algebras, the calculus relies on external sites to deal with non-concurrency issues. Many of the lower-level problems, such as management of locks and shared state, are delegated to sites in Orc. The Orc language was designed to provide a minimal base to experiment with the Orc combinators. Therefore, the language includes only the basic data types and structuring mechanisms. A site library provides additional capabilities for creating references, arrays, and channels, for example. Such a

combination has proved fruitful, as we have demonstrated in programming a classic example, Quicksort.

We hope that designers of future languages will adopt the fundamental principle we have espoused in the design of Orc: seamless integration of concurrency, structure and interaction with the external world.

Acknowledgements Jayadev Misra is deeply grateful to Tony Hoare for research ideas, inspiration and personal friendship spanning over three decades. It is no exaggeration that he would not have pursued certain research directions, that have ultimately proved quite successful, had it not been for Tony's encouragement. This paper is a small token of appreciation.

References

1. J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent programming in ERLANG (2nd ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996.
2. P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Communications of the ACM*, 14(10):667–668, 1971.
3. R. Fisher and F. Yates. *Statistical tables for biological, agricultural and medical research*. Oliver and Boyd, London, third edition, 1948.
4. C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. *International Joint Conference on Artificial Intelligence*, 1973.
5. C. A. R. Hoare. Partition: Algorithm 63, quicksort: Algorithm 64, and find: Algorithm 65. *Communications of the ACM*, 4(7):321–322, 1961.
6. C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.
7. D. Kitchin, W. R. Cook, and J. Misra. A language for task orchestration and its semantic properties. In *CONCUR*, pages 477–491, 2006.
8. D. Kitchin, A. Quark, W. Cook, and J. Misra. The Orc programming language. In D. Lee, A. Lopes, and A. Poetzsch-Heffter, editors, *Formal techniques for Distributed Systems; Proceedings of FMOODS/FORTE*, volume 5522 of *LNCS*, pages 1–25. Springer, 2009.
9. D. Kitchin, A. Quark, W. R. Cook, and J. Misra. Orc user guide. http://orc.csres.utexas.edu/userguide/html/index.html.
10. D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, third edition, 1997.
11. D. E. Knuth. *Sorting and Searching*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, second edition, 1998.
12. R. Milner. *A Calculus of Communicating Systems*. Springer Verlag, 1980.
13. R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, May 1999.
14. J. Misra. Computation orchestration: A basis for wide-area computing. In M. Broy, editor, *Proc. of the NATO Advanced Study Institute, Engineering Theories of Software Intensive Systems*, NATO ASI Series, Marktoberdorf, Germany, 2004.
15. L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.
16. A. Quark, D. Kitchin, W. R. Cook, and J. Misra. Orc language project website. http://orc.csres.utexas.edu.
17. R. Sedgewick. *Quicksort*. PhD thesis, Stanford University, 1975.
18. I. Wehrman, D. Kitchin, W. Cook, and J. Misra. A timed semantics of Orc. *Theoretical Computer Science*, 402(2-3):234–248, August 2008.
19. I. Wehrman, D. Kitchin, W. R. Cook, and J. Misra. Properties of the timed operational and denotational semantics of Orc. Technical Report TR-07-65, The University of Texas at Austin, Department of Computer Sciences, December 2007.