The Orc Programming Language

David Kitchin, Adrian Quark, William Cook, and Jayadev Misra

The University of Texas, Austin

Abstract. Orc was originally presented as a process calculus. It has now evolved into a full programming language, which we describe in this paper. The language has the structure and feel of a functional programming language, yet it handles many non-functional aspects effectively, including spawning of concurrent threads, time-outs and mutable state. We first describe the original concurrency combinators of the process calculus. Next we describe a small functional programming language that forms the core language. Then we show how the concurrency combinators of the process calculus and the functional core language are integrated seamlessly. The resulting language and its supporting libraries have proven very effective in describing typical concurrent computations; we demonstrate how several practical concurrent programming problems are easily solved in Orc.

1 Introduction

Concurrency has become an urgent practical problem with the advent of multi-core and multi-CPU systems. Unfortunately, concurrency is difficult for programmers, who are expected to define and manage threads and locks explicitly. These basic concurrency constructs do not serve well to express the high-level structure of control flow in concurrent programs, especially when handling failure, time-outs, and process termination. It is also important to easily compose processes at different granularities, from small processes to complete workflows, and to seamlessly integrate asynchronous human interactions within a process.

The Orc process calculus [8] was designed to express orchestrations and wide-area computations in a simple and structured manner. Its intent was to overcome some of the problems listed above. It is inherently concurrent and implicitly distributed. It has a clearly specified operational semantics. Human actors and computational processes of varying granularity are treated uniformly.

Orc was originally conceived as a formal model rather than a concrete tool. We have now implemented Orc as a complete programming language and begun writing substantial applications in it [11]. This paper describes the language and some of the motivating philosophy in its design. The language includes the three original concurrency combinators of Orc and a fourth one, introduced to detect terminations of computations. These combinators are seamlessly integrated with a core functional language. The complete language, called just Orc, has the structure and feel of a functional programming language; yet it handles many non-functional aspects effectively, including spawning of concurrent threads, time-outs and mutable state.

D. Lee et al. (Eds.): FMOODS/FORTE 2009, LNCS 5522, pp. 1–25, 2009.

The paper is structured as follows. In Section 2, we review the Orc concurrency calculus. In Section 3, we present the functional core of Orc. Section 4 integrates the functional core with the concurrency calculus, resulting in the full Orc programming language, which is subsequently enhanced with some helpful syntactic sugar, a library of useful services and functions, and the capability to interact with other languages such as Java. Section 5 presents a series of examples demonstrating solutions to common concurrent programming problems using the Orc language. We consider Orc's relationship to similar concurrent languages in Section 6, and conclude with some remarks on future work in Section 7.

We encourage the reader to visit our website [11] for more information. It hosts a comprehensive user guide, 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 concurrent *combinators*. When executed, an Orc expression calls services and may *publish* values. Different executions of the same expression may have completely different behaviors; they may call different services, receive different responses from the same service, and publish different values.

Orc expressions use *sites* to refer to external services. 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 present the calculus informally in this paper. The formal operational and denotational semantics of the calculus are given in [4].

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.

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.
`swap(`l_0`,`l_1`)`	Swap the values stored at locations l_0 and l_1.
`Weather("Austin, TX")`	Find the current weather in Austin.
`random(10)`	Get a random integer in the range 0..9.
`invertMatrix(`m`)`	Find the inverse of matrix m.

A site may give at most one response to a call. A site call may explicitly report that it will never respond, in which case we say that the call has *halted*.

For example, the call might represent an invalid operation (`invertMatrix` on a non-invertible matrix), it may have encountered a system error (trying to `swap` with a protected memory location), or it may simply have no available response (calling `CNN` on a future date). Some site calls may neither respond nor halt. For example a call to `Prompt` will wait forever if the user never inputs a response.

Though the Orc calculus itself contains no sites, there are a few fundamental sites which are so essential to writing useful computations 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 halts.

2.2 Combinators

Orc has four combinators to compose expressions: the parallel combinator `|`, the sequential combinator `>x>`, the pruning combinator[1] `<x<`, and the otherwise combinator[2] `;`.

When composing expressions, the `>x>` combinator has the highest precedence, followed by `|`, then `<x<`, and finally `;` with the lowest precedence.

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 these two computations.

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 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 values, no executions of G occur. 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)
```

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

[2] The `;` combinator is a new addition to Orc, inspired by practical experience writing Orc programs. Its formal semantics will appear in a forthcoming technical report.

The sequential combinator is left 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.

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

Otherwise Combinator. The execution of F `;` G first executes F. If F publishes no values and then *halts*, then G executes. We say that F halts if all of the following conditions hold:

1. All site calls in the execution of F have either responded or halted.
2. F will never call any more sites.
3. F will never publish any more values.

The following expression calls `CNN` to get the news for date d. If `CNN(`d`)` responds, `email` is called using that response, and the `BBC` is never called. However, the site `CNN` may halt, as described in Section 2.1, if there is no news available for date d. In this case `BBC(`d`)` is executed, and if it responds, `email` is called.

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

The otherwise combinator is associative, i.e. `(`F `;` G`)` `;` H is the same as F `;` `(`G `;` H`)`.

`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`** halts when executed; it is equivalent to `if(false)`.

2.3 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. A call $E(\overline{p})$ is evaluated by replacing the formal parameters $\overline{x}$ by the actual parameters $\overline{p}$ in the body F. Unlike a site call, a function call does not suspend if one of its arguments is a variable with no value. A function call may publish more than one value; it publishes every value published by the execution of F. Definitions may be recursive.

2.4 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].

The following example defines a metronome, which publishes a signal once every `t` milliseconds, indefinitely.

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

We can also use `Rtimer` together with the `<x<` combinator to enforce a timeout. For example, 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.")
```

3 Functional Core Language

In the preceding section, we introduced an abstract language intended to highlight several issues pertaining to concurrency. However, concurrent programs contain large amounts of sequential code, with attendant data and control structures. We enhance Orc by adding a functional core language to it. In this section we describe *Cor*, the functional core. In Section 4 we will show how any Cor expression can be written as an equivalent Orc expression; this will allow us to integrate Cor into Orc.

A Cor program is an expression. Cor expressions are built up recursively from smaller expressions. Cor evaluates an expression to reduce it to some simple value which cannot be evaluated further, for example a list of numbers or a Boolean truth value. This value is called the *result* of the expression. Some expressions do not have a result, because they represent an invalid computation (such as division by zero) or an infinite computation. Such expressions are called *silent*.

Values and Operators. Cor has three types of constants: numbers (5, -1, 2.71828, $\cdots$), strings (`"orc"`, `"ceci n'est pas une |"`, $\cdots$), and

[3] An implementation can only approximate this guarantee.

booleans (`true` and `false`). It provides the usual arithmetic (`+ - * / ···`), logical (`&& || ···`), and comparative (`= < > ···`) 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"`.

Conditionals. Cor has conditional expressions: **`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` `true` **`then`** `4` **`else`** `5`	evaluates to `4`.
`if` `0 < 5` **`then`** `0/5` **`else`** `5/0`	evaluates to `0`.
`if` `false || false` **`then`** `4+5` **`else`** `9/0`	is silent.

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 because Cor is a pure functional language and has no mutable state. If the same variable is bound again, subsequent references to that variable will use the new binding, but previous references remain unchanged. Declarations obey the rules of lexical scope.

What if a variable is bound to a silent expression? Consider:

```
val x = 1/0
val y = 4+5
if false then x else y
```

Any expression that evaluates `x` will be silent. However, the evaluation of **`val`** `y = 4+5` and **`if`** `false` **`then`** `x` **`else`** `y` proceeds as normal.

Data Structures. Cor supports two basic data structures: *tuples* (`(3, 7)`, `("tag", true, false)`, ···) and finite *lists* (`[4,4,1]`, `["example"]`, `[]`, ···). 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.

```
(1+3):[2+5,6]   evaluates to [4,7,6].
2:2:5:[]        evaluates to [2,2,5].

val t = [3,5]
1:t             evaluates to [1,3,5].

2:3             is silent, because 3 is not a list.
```

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

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

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

Functions. Functions are defined using the keyword **`def`**, in the following way.

```
def add(x,y) = x+y
```

The expression to the right of the `=` is called the *body* of the function.

The syntax for function calls is typical for functional programming. However, the interpretation of a function call is more elaborate, in anticipation of the concurrency combinators to be added. A call, such as `add(1+2,3+4)` is converted to a sequence of **`val`** declarations, one per formal parameter, followed by a goal expression, which is just the body of the function:

```
val x = 1+2
val y = 3+4
x+y
```

Notice that the evaluation strategy of functions allows a call to proceed even if some of the actual parameters are silent expressions, so long as the values of those actual parameters are not used in the evaluation of the body.

Functions may be defined recursively, as in:

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

Mutual recursion is also supported. There is no special keyword for mutual recursion; any contiguous sequence of function definitions is allowed to be mutually recursive.

Functions are actually values, just like any other. Defining a function creates a special value called a *closure*; the name of the function is a variable and its bound value is the closure. Since a closure is a value, it can be passed as an

argument to another function, thus allowing us to define a *higher-order* function. Cor functions obey *lexical closure*.

```
def onetwosum(f) = f(1) + f(2)
def triple(x) = x * 3
onetwosum(triple)
```

This is the same as `triple(1) + triple(2)`, i.e. `1 * 3 + 2 * 3`.

We may create a closure without giving it a name, using the keyword **lambda**. For example, `onetwosum( lambda(x) = x * 3 )` is equivalent to `onetwosum(triple)` as defined above.

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 sum([]) = 0
def sum(h:t) = h + sum(t)

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

4 The Orc Programming Language

The full Orc programming language combines the concurrency calculus with the functional core language, and adds to it a few notational conveniences and the support of a large library of sites and predefined functions. The full language has many features which we do not discuss here due to space constraints; see the Orc User Guide [5] for details.

4.1 Translating Cor to Orc

Every Cor expression can be translated to an equivalent expression in the Orc calculus, using a small set of primitive sites to perform tasks such as arithmetic and list manipulation.

Values and Operators. 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 which performs addition.

If a value or variable is mentioned alone, such as `3` or `x`, it becomes a call to the identity site `let` with that argument.

Cor expressions may be recursively nested. However, site calls in the Orc calculus may only have values or variables as arguments. For example, `2+(3+4)` cannot translate directly to `add(2, add(3,4))`, since the call `add(3,4)` cannot

be an argument. Instead, we translate `2+(3+4)` to `add(2,z) <z< add(3,4)`, where `z` is a fresh variable name. This translation can be applied to any such nested expression.

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

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

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

F `<x<` G

Data Structures. 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,t) <s< cons(2,t) <t< nil()`.

Patterns. 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. For example, the site `trycons` takes one argument; if that argument is a nonempty list, then it returns a tuple of the list's head and tail, otherwise it remains silent.

Functions. Cor function definitions translate to Orc definitions, though we must extend the calculus slightly to permit such definitions to occur within the scope of normal expressions and create closures.

The expression **`lambda`**`(...) =` E translates to the following Orc expression, where `f` is a fresh variable name:

`def` `f(...) =` E
`let(f)`

4.2 Integrating Cor and Orc

As we have seen, any Cor expression can be translated into an equivalent Orc expression. This allows us to integrate the concurrency combinators with Cor.

Mingling Cor and Orc expressions. The combined language allows any Cor or Orc expression to appear as a subexpression of any other Cor or Orc expression. Sometimes, an Orc expression which publishes multiple values will appear in the context of a Cor expression which expects only one value, for example `2 + (3 | 4)`. In such a case, the Orc expression executes until it publishes a value, and then is terminated. Therefore `2 + (3 | 4)` is to be understood as `(2 + x) <x< (3 | 4)`. While this may be an unexpected behavior, it is quite

useful once it becomes familiar. In fact, it is used so often in Orc programs that the pruning combinator itself is rarely written explicitly.

Patterns in Combinators. The combined language also allows patterns to 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.
```

Functions. A function call in Orc, as in Cor, binds the values of its actual parameters to its formal parameters, and then executes the function body with those bindings. Whenever the function body publishes a value, the function call publishes that value. Thus, unlike a site call, or a pure functional Cor call, an Orc function call may publish many values.

In the context of Orc, function calls are not strict. When a function call executes, it begins to execute the function body immediately, and also executes the argument expressions in parallel. When an argument expression publishes a value, it is terminated, and the corresponding formal parameter is bound to that value in the execution of the function body. Any part of the function body which uses a formal parameter that has not yet been bound suspends until that parameter is bound to a value.

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

This 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.

```
def throw() = random(6) + 1

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

Here is the translation of this program into the Orc calculus. Site `add` returns the sum of its arguments, `not` returns the negation of its boolean argument, and `equals` returns `true` iff its two arguments are equal.

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

def exp(n,c) =
   ( 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,c)
                       <p< add(q,r)
                         <q< throw()
                         <r< throw() )
         <y< ( exp(m,c) <m< sub(n,1) ) )
   ) <b< equals(n,0)
```

4.3 The . Notation

We will see many sites which represent objects with multiple behaviors. It is often convenient to treat such sites as if they had *methods*, in an object-oriented style. We access methods on sites using a special form of site call, as in `c.put(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. It 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)`.

4.4 Site Library

The Orc programming language has access to a library of useful sites. We introduce a few essential ones here.

Channels. Orc has no communication primitives like π-calculus channels [7] or Erlang mailboxes [1]. Instead, it makes use of sites to create channels of communication.

The most frequently used of these sites is `Buffer`. When called, it 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 waiting for a value if none is available. The call `c.put(v)` puts `v` as the last item of `c` and publishes a signal.

A channel may be closed to indicate that it will not be sent any more values. If the channel `c` is closed, `c.put(v)` always halts (without modifying the state of the channel), and `c.get()` halts once `c` becomes empty. The channel `c` may be closed by calling either `c.close()`, which returns a signal once `c` becomes empty, or `c.closenb()`, which returns a signal immediately.

References. Unlike imperative programming languages, 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` [9].

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 `r.read()` publishes the current value stored in `r`. If `r` is undefined, `r.read()` blocks until some write occurs.

Semaphores. Unlike other concurrent languages, Orc does not have any locking mechanisms built into the language. 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`, blocking if it cannot be acquired yet because its value is zero. The call `s.release()` releases `s`, increasing is value by one.

4.5 Function Library

The Orc programming language also includes a library of predefined functions. These are functions written in Orc, using **`def`**. We introduce a few important ones here.

`each`. One of the most common uses for a list is to send each of its elements through a sequential combinator. Since the list itself is a single value, we want to walk through the list and publish each one of its elements in parallel as a value. The library function `each` does exactly that.

Suppose we want to send the message `invite` to each email address in the list `inviteList`:

```
each(inviteList) >address> email(address, invite)
```

`repeat`. A site in Orc responds with at most one value. If we wish to model a service that outputs a stream of values, we can either encode the stream as a single value, or we can call the site repeatedly to receive successive values in the stream. The function `repeat` is very useful for the latter approach: `repeat(M)` calls site `M`, waits for it to respond, publishes the response, and then repeats the process, until the call to `M` halts. For example, `repeat(c.get)` will take and publish values from the channel `c` whenever they are available.

List Functions. Orc adopts many of the list idioms of functional programming. The Orc library contains definitions for most of the standard list functions, such as `map` and `fold`. Many of the list functions internally take advantage of concurrency to make use of any available parallelism; for example, the `map` function uses a fork-join to dispatch all of the mapped calls concurrently and assemble the result list from their responses.

4.6 Interacting with Other Languages

The current Orc implementation allows Java classes to be used as sites. A special declaration makes a Java class constructor available to Orc as a site with the same name. Calling that constructor site creates an instance of the Java object. That object's methods and fields can then be accessed using the dot notation described earlier. We anticipate that this mechanism may be generalized to other languages.

4.7 Complete Syntax

The complete abstract syntax of the Orc language used for the remainder of this paper is given below.

Constant	C	::=	boolean, number, or string
Variable	X	::=	variable name
Message	m	::=	message name
Pattern	P	::=	$C \mid X \mid (\overline{P}) \mid [\overline{P}] \mid \_$
Declaration	D	::=	**val** P = E
			| **def** $X(\overline{P})$ = E
Expression	E	::=	$C \mid X \mid (\overline{E}) \mid [\overline{E}] \mid$ **signal**
			| E *op* $E \mid X(\overline{E}) \mid X.m(\overline{E})$
			| **if** E **then** E **else** E
			| **lambda**$(\overline{P})$ = E
			| D E
			| E **|** E
			| E >P> E
			| E <P< E
			| E ; E
			| **stop**

5 Programming Idioms

In this section we give Orc implementations of some standard idioms from concurrent and functional programming. Despite the austerity of Orc's four combinators, we are able to encode a variety of idioms straightforwardly.

5.1 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 easy to express in Orc:

`(`F`, `G`)`

Recall that this is equivalent to:

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

This implementation 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 produce numbers and we wish to fork-join and add their results, we write simply F `+` G.

Example: Simple Parallel Auction. Orc programs often use fork-join 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 return the highest bid. The function `auction` performs this task (`max` finds the maximum of its arguments):

```
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 5.5 presents a different solution that addresses the issue of non-termination.

Example: Barrier Synchronization. Consider an expression of the following form, where F and G are expressions and `M` and `N` are sites:

`M() >x>` F `|` `N() >y>` G

Suppose we would like to *synchronize* F and G, so that both start executing at the same time, after both `M()` and `N()` respond. This is easily done using the fork-join idiom. In the following, we assume that `x` does not occur free in G, nor `y` in F.

`(M(), N()) >(x,y)> (` F `|` G `)`

5.2 Sequential Fork-Join

Previous sections illustrate how Orc can use the fork-join idiom to process a fixed set of expressions or a list of values. Suppose that instead we wish to process all the publications of an expression F, and once this processing is complete, execute some expression G. For example, F publishes the contents of a text file, one line at a time, and we wish to print each line to the console using the site `println`, then publish a signal after all lines have been printed.

Sequential composition alone is not sufficient, because we have no way to detect when all of the lines have been processed. A recursive fork-join solution would require that the lines be stored in a traversable data structure like a list, rather than streamed as publications from F. A better solution uses the `;` combinator to detect when processing is complete:

F `>x> println(x) >>` **`stop ; signal`**

Since ; only evaluates its right side if the left side does not publish, we suppress the publications on the left side using **stop**. Here, we assume that we can detect when F halts. If, for example, F is publishing the lines of the file as it receives them over a socket, and the sending party never closes the socket, then F never halts and no signal is published.

5.3 Priority Poll

The otherwise combinator is also useful for trying alternatives in sequence. Consider an expression of the form $F_0 \ ; \ F_1 \ ; \ F_2 \ ; \ \cdots$. If F_i does not publish and halts, then F_{i+1} is executed. We can think of the F_i's as a series of alternatives that are explored until a publication occurs.

Suppose that we would like to poll a list of buffers for available data. The list of buffers is ordered by priority. The first buffer in the list has the highest priority, so it is polled first. If it has no data, then the next buffer is polled, and so on.

Here is a function which polls a prioritized list of buffers in this way. It publishes the first item that it finds, removing it from the originating buffer. If all buffers are empty, the function halts. We use the `getnb` ("get non-blocking") method of the buffer, which retrieves the first available item if there is one, and halts otherwise.

```
def priorityPoll([]) = stop
def priorityPoll(b:bs) = b.getnb() ; priorityPoll(bs)
```

5.4 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 is silent.

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

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

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 can publish `true` immediately regardless of whether the other variable is available. Therefore we run `if(a) >> true` and `if(b) >> true` in parallel to wait for either variable

to become `true` and immediately publish the result `true`. Since more than one of these expressions may publish `true`, the surrounding `let(...)` is necessary to select and publish only the first result.

5.5 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. The following program runs F for at most one second, publishing its result if available and the value `0` otherwise.

```
let( F | Rtimer(1000) >> 0 )
```

Example: Auction with Timeout. The auction example in Section 5.1 may never finish if one of the bidders does not respond. We can add a timeout so that each bidder has at most 8 seconds to provide a bid:

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

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

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
```

Priority. We can use a timer 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 )
```

5.6 Metronome

A timer can be used to execute an expression repeatedly at regular intervals, for example to poll a service. Recall the definition of `metronome` from Section 2.4:

```
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 program is not defined recursively; the recursion is entirely contained within `metronome`.

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

5.7 Fold

We consider various concurrent implementations of the classic "list fold" function from functional programming:

```
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 a 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.

Associative Fold. We first consider the case when the reduction operator is associative. We define `afold(b,xs)` where `b` is a binary associative function and `xs` is a non-empty list. The implementation iteratively reduces `xs` to a single value. Each step of the iteration applies the auxiliary function `step`, which halves the size of `xs` by reducing disjoint pairs of adjacent items. Notice that `b(x,y):step(xs)` is an implicit fork-join, as described in Section 5.1. Thus, the call `b(x,y)` executes in parallel with the recursive call `step(xs)`, and as a result, all of the calls to `b` in each iteration occur in parallel.

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

Associative, Commutative Fold. We can devise a better strategy when the fold operator is both associative and commutative. We define `cfold(b,xs)`, where `b` is a binary associative and commutative function over two arguments 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`. 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(b, xs) =
  val c = Buffer()

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

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

  xfer(xs) | combine(length(xs))
```

5.8 Routing

The Orc combinators restrict the passing of values among their component expressions. However, some programs will require greater flexibility. For example, F `<x<` G provides F with the first publication of G, but what if F needs the first n publications of G? In cases like this we use channels or other stateful sites to redirect or store publications. We call this technique *routing* because it involves routing values from one execution to another.

Generalizing Termination. The pruning combinator terminates an expression after it publishes its first value. We have already seen how to use pruning just for its termination capability, without binding a variable, using the `let` site. Now we use routing to terminate an expression under different conditions, not just when it publishes a value; it may publish many values, or none, before being terminated.

Our implementation strategy is to route the publications of the expression through a channel so that we can put the expression inside a pruning combinator and still see its publications without those publications terminating the expression.

Enhanced Timeout. As a simple demonstration of this concept, we construct a more powerful form of timeout: allow an expression to execute, publishing arbitrarily many values (not just one), until a time limit is reached.

```
val c = Buffer()
repeat(c.get) <<
    F >x> c.put(x) >> stop
  | Rtimer(1000) >> c.closenb()
```

This program allows F to execute for one second and then terminates it. Each value published by F is routed through channel `c` so that it does not terminate

F. After one second, `Rtimer(1000)` responds, triggering the call `c.closenb()`. The call `c.closenb()` closes `c` and publishes a signal, terminating F. The library function `repeat` is used to repeatedly take and publish values from `c` until it is closed.

Interrupt. We can use routing to interrupt an expression based on a signal from elsewhere in the program. We set up the expression like a timeout, but instead of waiting for a timer, we wait for the semaphore `done` to be released. Any call to `done.release` will terminate the expression (because it will cause `done.acquire()` to publish), but otherwise F executes as normal and may publish any number of values.

```
val c = Buffer()
val done = Semaphore(0)
repeat(c.get) <<
    F >x> c.put(x) >> stop
  | done.acquire() >> c.closenb()
```

Publication Limit. We can use the interrupt idiom to limit an expression to n publications, rather than just one. Here is an expression that executes F until it publishes 5 values, and then terminates it.

```
val c = Buffer()
val done = Semaphore(0)
def allow(0) = done.release() >> stop
def allow(n) = c.get() >x> ( x | allow(n-1) )
allow(5) <<
    F >x> c.put(x) >> stop
  | done.acquire() >> c.closenb()
```

We use the auxiliary function `allow` to get only the first 5 publications from the channel `c`. When no more publications are allowed, `allow` uses the interrupt idiom to halt F and close `c`.

Non-Terminating Pruning. We can use routing to create a modified version of the pruning combinator. As in F `<x<` G, we'll run F and G in parallel and make the first value published by G available to F. However instead of terminating G after it publishes a value, we will continue running it, ignoring its remaining publications.

```
val r = Ref()
(F <x< r.read()) | (G >x> r.write(x))
```

Publication-Agnostic Otherwise. We can also use routing to create a modified version of the otherwise combinator. We'll run F until it halts, and then run G, regardless of whether F published any values or not.

```
val c = Buffer()
repeat(c.get) | (F >x> c.put(x) >> stop
                 ; c.close() >> G)
```

We use `c.close()` instead of the more common `c.closenb()` to ensure that G does not execute until all the publications of F have been routed. Recall that `c.close()` does not return until `c` is empty.

5.9 Larger Examples

In the previous sections we demonstrated how various concurrent programming idioms are expressed simply in Orc. Now we apply these idioms to solve non-trivial problems. The following examples illustrate how the different aspects of Orc — including the functional core, concurrency, time, synchronization and mutable state — combine to produce concise and efficient programs.

Dining Philosophers. The dining philosophers problem is a well known and intensely studied problem in concurrent programming. Five philosophers sit around a circular table. Each philosopher has two forks that she shares with her neighbors (giving five forks in total). Philosophers think until they become hungry. A hungry philosopher picks up both forks, one at a time, eats, puts down both forks, and then resumes thinking. Without further refinement, this scenario allows deadlock; if all philosophers become hungry and pick up their left-hand forks simultaneously, no philosopher will be able to pick up her right-hand fork to eat. Lehmann and Rabin's solution [6], which we implement, requires that each philosopher pick up her forks in a random order. If the second fork is not immediately available, the philosopher must set down both forks and try again. While livelock is still possible if all philosophers take forks in the same order, randomization makes this possibility vanishingly unlikely.

The following program gives the Orc implementation of dining philosophers. The `phil` function simulates a single philosopher. It takes as arguments two binary semaphores representing the philosopher's forks, and calls the `thinking`, `hungry`, and `eating` functions in a continuous loop. A `thinking` philosopher waits for a random amount of time, with a 10% chance of thinking forever. A `hungry` philosopher uses the `take` function to acquire two forks. An `eating` philosopher waits for a random time interval and then uses the `drop` function to relinquish ownership of her forks.

Calling `take` attempts to acquire a pair of forks `(a,b)` in two steps: wait for fork `a` to become available, then immediately attempt to acquire fork `b`. The call `b.acquirenb()` either acquires `b` and responds immediately, or halts if `b` is not available. If `b` is acquired, signal success; otherwise, release `a`, and then try again, randomly changing the order in which the forks are acquired using the auxiliary function `shuffle`.

The function `philosophers` recursively creates a chain of `n` philosophers, bounded by fork `a` on the left and `b` on the right. The goal expression of the program calls `philosophers` to create a chain of five philosophers bounded on the left and right by the same fork; hence, a ring.

```
def shuffle(a,b) = if (random(2) = 1) then (a,b) else (b,a)

def take((a,b)) =
  a.acquire() >> b.acquirenb() ;
  a.release() >> take(shuffle(a,b))

def drop(a,b) = (a.release(), b.release()) >> signal

def phil(a,b) =
  def thinking() =
    if (random(10) < 9)
      then Rtimer(random(1000))
      else stop
  def hungry() = take((a,b))
  def eating() =
    Rtimer(random(1000)) >>
    drop(a,b)
  thinking() >> hungry() >> eating() >> phil(a,b)

def philosophers(1,a,b) = phil(a,b)
def philosophers(n,a,b) =
  val c = Semaphore(1)
  philosophers(n-1,a,c) | phil(c,b)

val fork = Semaphore(1)
philosophers(5,fork,fork)
```

This Orc solution has several nice properties. The overall structure of the program is functional, with each behavior encapsulated in its own function, making the program easy to understand and modify. Mutable state is isolated to the "fork" semaphores and associated `take` and `get` functions, simplifying the implementation of the philosophers. The program never manipulates threads explicitly, but instead expresses relationships between activities using Orc's combinators.

Quicksort. The original quicksort algorithm [3] 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. We encode parts of the algorithm as concurrent activities where sequentiality is unneeded.

The following listing gives the implementation of the `quicksort` function which sorts the array `a` in place. The auxiliary function `sort` sorts the subarray given by indices `s` through `r` by calling `part` to partition the subarray and then recursively sorting the partitions.

The function `part` partitions the subarray given by indices `s` through `t` into two partitions, one containing values $\leq$ `p` and the other containing values $>$ `p`. The last index of the lower partition is returned. The value at `a(s)` is assumed to be $\leq$ `p` — this is satisfied by choosing `p = a(s)?` initially. To create the partitions, `part` calls two auxiliary functions `lr` and `rl` concurrently. These functions scan from the left and right of the subarray respectively, looking for out-of-place elements. Once two such elements have been found, they are swapped using the auxiliary function `swap`, and then the unscanned portion of the subarray is partitioned further. Partitioning is complete when the entire subarray has been scanned.

This program uses the syntactic sugar `x?` for `x.read()` and `x := y` for `x.write(y)`. Also note that the expression `a(i)` returns a reference to the element of array `a` at index `i`, counting from 0.

```
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)
```

Meeting Scheduler. Orc makes very few assumptions about the behaviors of services it uses. Therefore it is straightforward to write programs which interact with human agents and network services. This makes Orc especially suitable for encoding *workflows* [2], the coordination of multiple activities involving multiple participants. The following program illustrates a simple workflow for scheduling a business meeting. Given a list of people and a date range, the program asks each person when they are available for a meeting. It then combines all the responses, selects a meeting time which is acceptable to everyone, and notifies everyone of the selected time.

```
val during = DateTimeRange(LocalDate(2009, 9, 10),
                           LocalDate(2009, 10, 17))
val invitees = ["dkitchin@cs.utexas.edu", "quark@cs.utexas.edu"]

def invite(invitee) =
  Form() >f>
  f.addPart(DateTimeRangesField("times",
    "When are you available for a meeting?", during, 9, 17)) >>
  f.addPart(Button("submit", "Submit")) >>
  SendForm(f) >receiver>
  SendMail(invitee, "Meeting Request", receiver.getURL()) >>
  receiver.get() >response>
  response.get("times")

def notify([]) =
  each(invitees) >invitee>
  SendMail(invitee, "Meeting Request Failed",
                    "No meeting time found.")
def notify(first:_) =
  each(invitees) >invitee>
  SendMail(invitee, "Meeting Request Succeeded",
                    first.getStart())

map(invite, invitees) >responses>
afold(lambda (a,b) = a.intersect(b), responses) >times>
notify(times)
```

This program begins with declarations of `during` (the date range for the proposed meeting) and `invitees` (the list of people to invite represented by email addresses).

The `invite` function obtains possible meeting times from a given invitee, as follows. First it uses library sites (`Form`, `DateTimeRangesField`, `Button`, and `SendForm`) to construct a web form which may be used to submit possible meeting times. Then it emails the URL of this form to the invitee and blocks waiting for a response. When the invitee receives the email, he or she will use a web browser to visit the URL, complete the form, and submit it. The corresponding execution of `invite` receives the response in the variable `response` and extracts the chosen meeting times.

The `notify` function takes a list of possible meeting times, selects the first meeting time in the list, and emails everyone with this time. If the list of possible meeting times is empty, it emails everyone indicating that no meeting time was found.

The goal expression of the program uses the library function `map` to apply `notify` to each invitee and collect the responses in a list. It then uses the library function `afold` to intersect all of the responses. The result is a set of meeting times which are acceptable to everyone. Finally, `notify` is called to select one of these times and notify everyone of the result.

This program may be extended to add more sophisticated features, such as a quorum (to select a meeting as soon as some subset of invitees responds) or timeouts (to remind invitees if they don't respond in a timely manner). These modifications are local and do not affect the overall structure of the program. For complete details, see examples on our website [11].

6 Related Work

Many attempts have been made to rethink the prevailing concurrency model of threads with shared state and put forth programming languages based on novel approaches.

Erlang is perhaps the most widely adopted of these languages [1]. It is founded on an actor model of concurrency, wherein sequential processes communicate via message-passing. While the Orc language is very different in design, Erlang has often served as a basis for comparison.

Oz is another novel concurrent programming language [12]. Though it uses a more conventional thread-based approach to concurrent computation, its model of shared state is much more structured and safe than that of mainstream languages. Rather than reducing all programs to a small calculus, Oz instead starts with a kernel calculus and incrementally expands it with new concepts that provide additional expressive capability, such as ports, objects, and computation spaces. While Oz is more structured than a pure message-passing model and its unification-based assignment operation is more expressive than Orc's pattern matching, evaluation of Oz programs requires access to a global unification store, which Orc programs do not need. The site `Cell` in the Orc library, which creates write-once mutable cells, was inspired by Oz's single-assignment variables.

Pict is a concurrent functional language [10] which is based on the π-calculus, in much the same way that the Orc language is based on the Orc calculus. In fact, Pict was an inspiration for some of the design choices of the Orc language. The primary difference between Pict and Orc is that Pict imposes a functional, continuation-passing structure on an unstructured communication calculus, whereas the Orc calculus is already structured, so a translation into the calculus preserves much of the structure of the original program and may thereby ease formal analysis.

7 Conclusion

We have presented Orc, developed from a simple concurrency calculus into a complete programming language capable of addressing real-world problems while maintaining its original formal simplicity.

Future Work. The language continues to be actively developed; current areas of development include a static type system with partial type inference, an explicit exception handling mechanism, a module system for namespace management and separate compilation, support for atomic sections using transactions, and dynamic modification of programs.

Acknowledgements. We would like to thank Andrew Matsuoka and John Thywissen for helpful discussions about the design of the Orc language.

References

1. Armstrong, J., Virding, R., Wikström, C., Williams, M.: Concurrent programming in ERLANG, 2nd edn. Prentice Hall International (UK) Ltd., Hertfordshire (1996)
2. Cook, W.R., Patwardhan, S., Misra, J.: Workflow patterns in Orc. In: Ciancarini, P., Wiklicky, H. (eds.) COORDINATION 2006. LNCS, vol. 4038, pp. 82–96. Springer, Heidelberg (2006)
3. Hoare, C.A.R.: Partition: Algorithm 63, quicksort: Algorithm 64, and find: Algorithm 65. Communications of the ACM 4(7), 321–322 (1961)
4. Kitchin, D., Cook, W.R., Misra, J.: A language for task orchestration and its semantic properties. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 477–491. Springer, Heidelberg (2006)
5. Kitchin, D., Quark, A., Cook, W.R., Misra, J.: Orc user guide, `http://orc.csres.utexas.edu/userguide/html/index.html`
6. Lehmann, D.J., Rabin, M.O.: On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers problem. In: POPL, pp. 133–138 (1981)
7. Milner, R.: Communicating and Mobile Systems: the π-Calculus. Cambridge University Press, Cambridge (1999)
8. Misra, J.: Computation orchestration: A basis for wide-area computing. In: Broy, M. (ed.) Proc. of the NATO Advanced Study Institute, Engineering Theories of Software Intensive Systems, NATO ASI Series, Marktoberdorf, Germany (2004)
9. Paulson, L.C.: ML for the Working Programmer. Cambridge University Press, Cambridge (1991)
10. Pierce, B.C., Turner, D.N.: Pict: A programming language based on the pi-calculus. In: Plotkin, G., Stirling, C., Tofte, M. (eds.) Proof, Language and Interaction: Essays in Honour of Robin Milner, pp. 455–494. MIT Press, Cambridge (2000)
11. Quark, A., Kitchin, D., Cook, W.R., Misra, J.: Orc language project website, `http://orc.csres.utexas.edu`
12. Van Roy, P., Haridi, S.: Concepts, Techniques, and Models of Computer Programming. The MIT Press, Cambridge (2004)