Orc User Guide v2.0.1

Orc User Guide v2.0.1

Publication date 2011-04-27

License and Grant Information

This material is based upon work supported by the National Science Foundation under Grant No. CCF-0811536. Any opinions, findings, and conclusions or recommendations expressed in this document are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Table of Contents

Preface v
1. An Introduction to Orc 1
    1.1. Simple Expressions 2
        1.1.1. Values 2
        1.1.2. Operators 2
        1.1.3. Sites 2
    1.2. Complex Expressions 4
        1.2.1. Combinators 4
        1.2.2. `val` 6
        1.2.3. Conditional Expressions 7
        1.2.4. Nested Expressions 7
    1.3. Data Structures 9
        1.3.1. Tuples 9
        1.3.2. Lists 9
        1.3.3. Patterns 10
    1.4. Functions 11
        1.4.1. Recursion 11
        1.4.2. Clauses 12
        1.4.3. Guards 13
2. Additional Features of Orc 14
    2.1. First-Class Functions 15
    2.2. The `.` notation 16
    2.3. Records 17
    2.4. Defining Sites in Orc 18
        2.4.1. Example: Stack 18
        2.4.2. Example: Multicast 18
    2.5. Datatypes 21
    2.6. Importing Resources 22
        2.6.1. Writing Custom Sites 22
        2.6.2. Using Java Classes 22
        2.6.3. Including Source Files 22
    2.7. Type Checking 23
        2.7.1. Type Inference 23
        2.7.2. Adding Type Information 23
        2.7.3. Polymorphism 24
3. Syntactic and Stylistic Conventions 26
    3.1. Parallel combinator 27
    3.2. Sequential combinator 28
    3.3. Pruning combinator 29
    3.4. Declarations 30
        3.4.1. Ambiguous Declarations 30
4. Programming Idioms 31
    4.1. Channels 32
    4.2. Lists 33
    4.3. Streams 34
    4.4. Mutable References 35
        4.4.1. Rewritable references 35
        4.4.2. Write-once references 35
        4.4.3. Syntax for manipulating references 36
        4.4.4. Arrays 36
    4.5. Tables 39

4.6. Loops 40
4.7. Parallel Matching 41
4.8. Fork-join 43
4.8.1. Example: Machine initialization 43
4.8.2. Example: Simple parallel auction 43
4.8.3. Example: Barrier synchronization 43
4.9. Sequential Fork-Join 45
4.10. Priority Poll 46
4.11. Parallel Or 47
4.12. Timeout 48
4.12.1. Auction with timeout 48
4.12.2. Detecting timeout 48
4.13. Priority 50
4.14. Metronome 51
4.15. Routing 52
4.15.1. Generalizing Termination 52
4.15.2. Non-Terminating Pruning 53
4.15.3. Sequencing Otherwise 53
4.16. Fold 55
4.16.1. Associative Fold 55
4.16.2. Associative, Commutative Fold 55
5. Larger Examples 57
5.1. Dining Philosophers 58
5.2. Hygienic Dining Philosophers 60
5.3. Readers-Writers 64
5.4. Quicksort 66
5.5. Meeting Scheduler 68

Preface

Orc is a programming language designed to make distributed and concurrent programs simple and intuitive to write. Orc expresses orchestration, a type of structured concurrency. It emphasizes the flow of control and gives a global view of a concurrent system. Orc is well-suited for task orchestration, a form of concurrent programming with applications in workflow, business process management, and web service orchestration. Orc provides constructs to orchestrate the concurrent invocation of services while managing time-outs, priorities, and failures of services or communication. To learn more about Orc and run your own Orc programs, visit the Web site: `http://orc.csres.utexas.edu/`.

Unless otherwise noted, all material in this document pertains to the Orc language implementation version 2.0.1 .

This guide introduces the reader to the Orc programming language. It does not cover every detail of the language; see the Reference Manual for comprehensive and authoritative documentation of all language features.

Chapter 1 introduces the essential concepts of Orc, such as communication with external services, and building complex orchestrations from simpler programs using the four concurrency combinators. Data structures and function definitions are also discussed.

Chapter 2 discusses some additional features of Orc that extend the basic syntax. These are useful for creating practical Orc programs, but they are not essential to the understanding of the language.

Chapters 3 through 5 turn our attention to how the language is used in practice, with guidelines on style and programming methodology, explanations of some common concurrency patterns, and larger example programs.

Chapter 1. An Introduction to Orc

An Orc program is an *expression*. Complex Orc expressions are built up recursively from simpler expressions. Orc expressions are *executed*; an execution may interact with external services, and *publish* some number of values (possibly zero). Publishing a value is similar to returning a value with a `return` statement in an imperative language, or evaluating an expression in a functional language, except that an execution may publish many times, at different times, or might not publish at all. An expression which does not publish is called *silent*.

An execution *halts* when it is finished; it will not interact with any more services, publish any more values, or have any other effects.

Different executions of the same expression may have completely different behaviors; they may call different services, may receive different responses from the same site, and may publish different values.

1.1. Simple Expressions

This section shows how to write some simple Orc expressions. Simple expressions publish at most one value, and do not recursively contain other expressions. We will see later how some of these cases may also be used as complex expressions.

1.1.1. Values

The simplest expression one can write is a literal value. Executing that expression simply publishes the value.

Orc has four kinds of literal values:

- Booleans: `true` and `false`
- Numbers: `5, -1, 2.71828, ...`
- Strings: `"orc"`, `"ceci n'est pas une |"`
- A special value `signal`.

1.1.2. Operators

Orc has a standard set of arithmetic, logical, and comparison operators. As in most other programming languages, they are written in the usual infix style. They have Java-like operator precedence, which can be overridden by adding parentheses.

Examples

- `1 + 2` publishes `3`.
- `(98 + 2) * 17` publishes `1700`.
- `4 = 20 / 5` publishes `true`.
- `3-5 >= 5-3` publishes `false`.
- `true && (false || true)` publishes `true`.
- `"leap" + "frog"` publishes `"leapfrog"`.
- `3 / 0` halts, publishing nothing.

1.1.3. Sites

An Orc program interacts with the external world by calling *sites*. Sites are one of the two fundamental concepts of Orc programming, the other being combinators which we discuss later when covering complex expressions.

A site call in Orc looks like a method, subroutine, or function call in other programming languages. A site call might publish a useful value, or it might just publish a `signal`, or it might halt, refusing to publish anything, or it might even wait indefinitely. Here are some examples:

Examples

- `Println("hello world")` prints `hello world` to the console and publishes a `signal`.

- `Random(10)` publishes a random integer from 0 to 9, uniformly distributed.
- `Browse("http://orc.csres.utexas.edu/")` opens a browser window pointing to the Orc home page and publishes a `signal`.
- `Error("I AM ERROR")` reports an error message on the console, and halts. It publishes nothing.
- `Rwait(420)` waits for 420 milliseconds, then publishes a `signal`.
- `Prompt("Username:")` requests some input from the user, then publishes the user's response as a string. If the user never responds, the site waits forever.

Even the most basic operations in Orc are sites. For example, all of the operators are actually sites; `2+3` is just another way of writing the site call `(+)(2,3)`.

By convention, all site names begin with a capital letter.

1.2. Complex Expressions

Complex expressions recursively contain other expressions. They may be formed in a number of ways: using one of Orc's four combinators, adding a declaration, adding a conditional expression, or using an expression as an operand or site call argument.

1.2.1. Combinators

The concurrency combinators are one of the two fundamental concepts of Orc programming, the other being sites. They provide the core orchestration capabilities of Orc: parallel execution, sequential execution, blocking on future values, terminating a computation, and trying an alternative if some computation halts.

1.2.1.1. Parallel

Orc's simplest combinator is `|`, the parallel combinator. Execution of the complex expression F `|` G, where F and G are Orc expressions, executes F and G concurrently. Whenever a value is published during the execution of F or G, the execution of F `|` G publishes that value. Note the publications of F and G are interleaved arbitrarily.

```
{- Publish 1 and 2 in parallel -}

1 | 1+1

{-
OUTPUT:PERMUTABLE
1
2
-}
```

The brackets `{-  -}` enclose *comments*, which are present only for documentation and are ignored by the compiler.

1.2.1.2. Sequential

Now that we have expressions which publish multiple values, what can we do with those publications? The sequential combinator, written F `>x>` G, combines the expression F, which may publish some values, with another expression G, which will use the values as they are published; the *variable* `x` transmits the values from F to G.

The execution of F `>x>` G starts by executing F. Whenever F publishes a value, a new execution of G begins in parallel with F (and with other executions of G). In that instance of G, variable `x` is bound to the value published by F. Values published by the executions of G are published by the whole expression, but the values published by F are not published by the whole expression; they are consumed by the variable binding.

```
{- Publish 1 and 2 in parallel -}

(0 | 1) >n> n+1

{-
OUTPUT:PERMUTABLE
1
2
-}


{- Publish 3 and 4 in parallel -}

2 >n> (n+1 | n+2)

{-
OUTPUT:PERMUTABLE
3
4
-}


{- Publish 5 -}

2 >x> 3 >y> x+y

{-
OUTPUT:
5
-}
```

The sequential combinator may also be written without a variable, as in F `>>` G. This has the same behavior, except that no variable name is given to the values published by F. When F publishes only one value, this is similar to a sequential execution in an imperative language. For example, suppose we want to print three messages in sequence:

```
{- Print three messages in sequence -}

Println("Yes") >>
Println("We") >>
Println("Can") >>
stop

{-
OUTPUT:PERMUTABLE
Yes
We
Can
-}
```

The simple expression `stop` does nothing and halts immediately. In conjunction with `>>`, it can be used to ignore unneeded publications, such as the `signal` that would be published by `Println("Can")`.

1.2.1.3. Pruning

The pruning combinator, written F `<x<` G, allows us to block a computation waiting for a result, or terminate a computation. The execution of F `<x<` G starts by executing F and G in parallel. Whenever F publishes a value, that value is published by the entire execution. When G publishes its first value, that value is bound to `x` in F, and then the execution of G is immediately *killed*. A killed expression cannot make any more site calls or publish any values.

During the execution of F, any part of the execution that depends on `x` will be blocked until `x` is bound. If G never publishes a value, parts of F may be blocked forever.

```
{- Publish either 5 or 6, but not both -}

x+2 <x< (3 | 4)

{-
OUTPUT:
5
-}
{-
OUTPUT:
6
-}
```

Though a terminated execution may not make any new calls, the calls that it has already made will continue normally; their responses are simply ignored. This may have surprising consequences when a call has side effects, as in the following example.

```
{- This example might actually print both "uh" and "oh" to the
   console, regardless of which call responds first. -}

stop <x< Println("uh") | Println("oh")

{-
OUTPUT:PERMUTABLE
uh
oh
-}
{-
OUTPUT:
uh
-}
{-
OUTPUT:
oh
-}
```

Both of the `Println` calls could be initiated before either one of them publishes a value and terminates the expression. Once the expression is terminated, no new calls occur, but the other `Println` call still proceeds and still has the effect of printing its message to the console.

1.2.1.4. Otherwise

Orc's fourth concurrency combinator, the *otherwise* combinator, is written F `;` G. The execution of F `;` G proceeds as follows. First, F is executed. If F halts, and has not published any values, then G executes. If F did publish one or more values, then G is ignored.

1.2.2. `val`

An expression may be preceded by one or more *declarations*. Declarations are used to bind values to be used in that expression (or *scope*).

The declaration `val x =` G, followed by expression F, executes G, and binds its first publication to `x`, to be used in F.

This is actually just a different way of writing the expression F `<x<` G. Thus, `val` shares all of the behavior of the pruning combinator: F executes in parallel with G, uses of `x` block until G has published, and when G publishes, it is killed. In fact, the `val` form is used much more often than the `<x<` form, since it is usually easier to read.

1.2.3. Conditional Expressions

Orc has a conditional expression, written `if` E `then` F `else` G. The `else` branch is required. Execution of `if` E `then` F `else` G first executes E. If E publishes `true`, E is terminated and F executes. If E publishes `false`, E is terminated and G executes.

1.2.4. Nested Expressions

The execution of an Orc expression may publish many values. What if we want to use such an expression in a context where only one value is expected? For example, what does `2 + (3 | 4)` publish?

Whenever an Orc expression appears in such a context, it executes until it publishes its first value, and then it is terminated. The published value is then used in the context. This allows any expression to be used as an operand of an operator expression or an argument to a site call.

```
{- Publish either 5 or 6 -}

2 + (3 | 4)

{-
OUTPUT:
5
-}
{-
OUTPUT:
6
-}
```

```
{- Publish exactly one of 0, 1, 2 or 3 -}

(0 | 2) + (0 | 1)

{-
OUTPUT:
0
-}
{-
OUTPUT:
1
-}
{-
OUTPUT:
2
-}
```

```
{-
OUTPUT:
3
-}
```

To be precise, whenever an Orc expression appears in such a context, it is treated as if it were on the right side of a pruning combinator, using a fresh variable name to fill in the hole. This is called deflation.

1.3. Data Structures

Orc supports three basic data structures, *tuples*, *lists*, and *records*. This section describes tuples and lists; records are described in a subsequent chapter.

1.3.1. Tuples

A *tuple expression* is a comma-separated sequence of at least two expressions, enclosed by parentheses. Each expression is executed and its first published value is taken; the value of the whole tuple expression is a tuple containing each of these values in order. If any of the expressions is silent, then the whole tuple expression is silent.

Examples

- `(1+2, 7)` evaluates to `(3,7)`.
- `("true" + "false", true || false, true && false)` evaluates to `("truefalse", true, false)`.
- `(2/2, 2/1, 2/0)` is silent, since `2/0` is a silent expression.

1.3.2. Lists

A *list expression* is a comma-separated sequence of expressions enclosed by square brackets. It may be of any length, including zero. Each expression is executed and its first published value is taken; the value of the whole list expression is a list containing each of these values in order. If any of the expressions is silent, then the whole list expression is silent.

Examples

- `[1,2+3]` publishes `[1,5]`.
- `[true && true]` publishes `[true]`.
- `[]` just publishes `[]`, the empty list.
- `[5, 5/0, 5]` is silent, since `5/0` is a silent expression.

There is also a concatenation (*cons*) operation on lists, written `F:G`, where F and G are expressions. It publishes a new list whose first element is the value of F and whose remaining elements are the list value of G.

Examples

- `(1+3):[2+5,6]` publishes `[4,7,6]`.
- `2:2:5:[]` publishes `[2,2,5]`.
- Suppose `t` is bound to `[3,5]`. Then `1:t` publishes `[1,3,5]`.
- `2:3` is silent, because `3` is not a list.

1.3.3. Patterns

The Orc language provides *pattern matching* capabilities to inspect and extract pieces of a data structure. A *pattern* may be used wherever a variable could be bound; for example, in a `val` declaration, or a sequential combinator.

The following `val` declarations bind `z` to `(3,4)`, `x` to `3`, and `y` to `4`, using the tuple pattern `(x,y)`:

```
val z = (3,4)
val (x,y) = z
```

The wildcard pattern `_` lets us ignore irrelevant parts of a data structure. Here is an expression which extracts the first element of a list:

```
[1,2,3] >first:_> first
```

Notice that cons (`:`) is being used as a pattern to separate a list into its head and tail.

A pattern may also be used as a filter. A literal value may be used as a pattern; if the same value is not present in the data structure being matched, the pattern will *fail*, and the value will be discarded. Here is an expression which publishes the second element of pairs with first element 0, and ignores any other pair:

```
( (0,3) | (1,4) | (2,5) | (0,6) ) >(0,x)> x
```

1.4. Functions

Like most other programming languages, Orc provides the capability to define *functions*, which are expressions that have a defined name, and have some number of *parameters*. Functions are declared 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. `x` and `y` are the parameters. By convention, all function names begin with a lowercase letter.

After defining the function, we can call it. A function call looks just like a site call. To execute a call, we treat it like a sequence of `val` declarations associating the parameters with the arguments, followed by the body of the function. Every value published by the body expression is published by the call. This is unlike a site call, which publishes at most once.

```
{- add(1+2, 3+4) is equivalent to: -}

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

{-
OUTPUT:
10
-}
```

Examples

- `add(10,10*10)` publishes `110`.
- `add(add(5,3),5)` publishes `13`.

Notice that the execution of a function call can proceed even if some of the arguments haven't published a value yet. The parts of the body that depend on them will simply block.

```
def demo(x,y) = x | y | x+y
demo(3, Rwait(2000) >> 4)
```

This call publishes `3` immediately, but blocks for 2 seconds before publishing `4` and `7`.

A function definition or call may have zero arguments, in which case we write `()` for the arguments.

```
def Zero() = 0
```

1.4.1. Recursion

A function can be recursive; that is, the name of a function may be used in its own body.

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

The call `sumto(5)` publishes 15.

A recursive function may run forever, publishing an infinite number of times. The function `metronome` is a classic example; a call to `metronome` publishes a `signal` once per second, forever:

```
def metronome() = signal | Rwait(1000) >> metronome()
```

Mutual recursion is also supported.

```
def even(n) =
  if (n :> 0) then odd(n-1)
  else if (n <: 0) then odd(n+1)
  else true
def odd(n) =
  if (n :> 0) then even(n-1)
  else if (n <: 0) then even(n+1)
  else false
```

There is no special keyword for mutual recursion; any contiguous sequence of function declarations is assumed to be mutually recursive. Also, note that `:>` and `<:` are the Orc symbols for 'greater than' and 'less than' respectively.

1.4.2. Clauses

The combination of functions and pattern matching offers a powerful capability: *clausal* definition of functions. We can define expressions which execute different code depending on the structure of their arguments.

Here's an example.

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

`sum(l)` publishes the sum of the numbers in the list `l`. It has two clauses: one which matches the empty list, and one which matches any nonempty list. If its argument is an empty list, it returns 0, the appropriate sum for an empty list. If the argument is a nonempty list, it adds the first element of that list to the sum of all of the other elements. In this way, it recursively finds the sum of the list.

A function may have multiple clauses, each of which has a sequence of patterns to match each argument, and a body expression. Naturally, all clauses of a function must have the same number of arguments. Any contiguous sequence of definitions with the same name and different arguments is interpreted as a clausal definition, where each individual declaration is a clause of the larger function.

When the function is called, the clauses are tried in the order in which they appear until a match is found. If no clause matches, the call remains silent.

We can use a literal pattern to define the base case of a recursive function in a straightforward way.

```
{- Fibonacci numbers -}
def fib(0) = 1
def fib(1) = 1
def fib(n) = fib(n-1) + fib(n-2)
```

This definition of the Fibonacci function is straightforward, but slow, due to the repeated work in recursive calls to `fib`. We can define a linear-time version, again with the help of pattern matching:

```
{- Alternate definition of the Fibonacci function -}

{- A helper function: find the pair (Fibonacci(n), Fibonacci(n+1)) -}
def H(0) = (1,1)
def H(n) = H(n-1) >(x,y)> (y, x+y)

def fib(n) = H(n) >(x,_)> x
```

As a more complex example of matching, consider the following function which takes a list argument and returns a new list containing only the first `n` elements of the argument list.

```
def take(0,_) = []
def take(n,h:t) = h:take(n-1, t)
```

1.4.3. Guards

Each clause of a function definition may also have a *guard*: a Boolean expression which determines whether or not the clause applies. If the guard publishes `false`, then the next clause is tried, just as if some pattern had failed to match.

We can add guards to a previous example to protect against the degenerate case of a negative argument:

```
{- Fibonacci numbers -}
def fib(0) = 1
def fib(1) = 1
def fib(n) if (n :> 1) = fib(n-1) + fib(n-2)
```

We can also improve the readability of a previous example:

```
def even(n) if (n :> 0) = odd(n-1)
def even(n) if (n <: 0) = odd(n+1)
def even(0) = true
def odd(n) if (n :> 0) = even(n-1)
def odd(n) if (n <: 0) = even(n+1)
def odd(0) = false
```

Chapter 2. Additional Features of Orc

Chapter 1 presented the essential features of the Orc programming language. Here we show a number of additional features, which are useful for programming in certain styles, making use of additional services and resources, and constructing larger programs.

2.1. First-Class Functions

In the Orc programming language, functions are first-class values. This means that a function is treated like any other value; it may be published, passed as an argument to a call, incorporated into a data structure, and so on.

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. For example, these function declarations create two closures, bound to the variables `a` and `b`, from which we subsequently create a tuple called `funs`:

```
def a(x) = x-3
def b(y) = y*4
val funs = (a,b)
```

A closure can be passed as an argument to another function. A function which accepts functions as arguments is called a *higher-order* function. Here's an example:

```
def diff(f) = f(1) - f(0)
def triple(x) = x * 3

diff(triple) {-  equivalent to triple(1) - triple(0)  -}
```

The use of higher-order functions is common in functional programming. Here is the Orc version of the classic 'map' function:

```
def map(f, []) = []
def map(f, h:t) = f(h):map(f,t)
```

Sometimes one would like to create a closure directly, without bothering to give it a name. There is a special keyword `lambda` for this purpose. By writing a function definition without the keyword `def` and replacing the function name with the keyword `lambda`, that definition becomes an expression which evaluates to a closure.

```
def diff(f) = f(1) - f(0)


diff( lambda(x) = x * 3 )
{-
  this is identical to:

  def triple(x) = x * 3
  diff(triple)
-}
```

2.2. The `.` notation

In many object-oriented programming languages, one calls a method or accesses a field of an object using the dot operator; for example, `obj.m()` calls the method `m` of the object `obj`.

There is a special kind of site call in Orc which serves a similar purpose. One may write `x.msg`, for any identifiers `x` and `msg`. This treats the value bound to `x` as a site, and calls it with a special *message* value `msg`. If the site understands the message `msg` (for example, if `x` is bound to a Java object with a field called `msg`), the site interprets the message and responds with some appropriate value. If the site does not understand the message sent to it, it does not respond, and no publication occurs. If `x` cannot be interpreted as a site, no call is made.

Typically this capability is used so that sites may be syntactically treated like objects, with multiple methods and fields. For example, a channel `c` might understand the messages `get` and `put`, to get values from and put values on that channel, respectively. Such calls would be written `c.get()`, or `c.put(6)`.

A call such as `c.put(6)` actually occurs in two steps. First `c.put` sends the message `put` to the site `c`; this publishes a site whose only purpose is to put values on the channel. Next, that site is called on the argument `6`, sending 6 on the channel. Readers familiar with functional programming will recognize this technique as *currying*.

2.3. Records

In addition to tuples and lists, the Orc language has a third native data structure, called a *record*.

A *record expression* is a comma-separated sequence of elements of the form k `=` E, enclosed by record braces `{.` and `.}`, where each k is an identifier called a key, and each E is an expression. Records may have any number of fields, including zero. Each expression is executed and its first published value is taken; the value of the whole record expression is a record containing a pairing of each key with its associated value. Order is irrelevant. If any of the expressions are silent, then the whole record expression is silent.

Examples

- `{. zero = 3 - 3, one = 0 + 1 .}` publishes `{. zero = 0, one = 1 .}`.
- `{. .}` publishes `{. .}`, the empty record.

Elements of records are accessed using the dot (`.`) syntax described earlier. The expression `r.k` publishes the value paired with key `k` in record `r`. If `k` is not present in `r`, the expression is silent.

Suppose `r = {. x = 0, y = 1 .}`

Examples

- `r.x` publishes `0`.
- `r.y` publishes `1`.
- `r.z` is silent.

Like tuples and lists, records can also be matched by a pattern. However, unlike other patterns, a record pattern does not need to name all of the keys in the record being matched; it only needs to match a subset.

Suppose `r = {. x = 0, y = 1, z = 2 .}`

Examples

- `r >{. y = a, x = b .}> (a,b)` publishes `(1,0)`.
- `r >{. y = a, w = b .}> (a,b)` is silent.

2.4. Defining Sites in Orc

Orc has a special declaration, `def class`, which allows a site to be written in Orc itself. This is a convenient mechanism for writing sites whose internal behavior is best expressed by an orchestration, which might be an awkward task in Java, Scala, or other similar languages.

A `def class` resembles a `def` in its syntax, but not in its behavior. The site defined by `def class` is a factory which creates instances of the class. Each `def` within the body of the `def class` becomes a method of an instance. The scope expression of these declarations becomes the internal computation of the instance; it begins to run when the instance is created, and cannot be killed by the Orc program, exactly as if it were a computation of an external service.

The following two examples demonstrate some of the behavior of `def class`. For a more comprehensive explanation of `def class` and its features, see the Reference Manual.

2.4.1. Example: Stack

The following code defines a site `Stack`, which creates stacks with `push` and `pop` methods:

```
def class Stack() =
  {- The stack is initially empty -}
  val store = Ref([])

  def push(x) =
    store? >xs>
    store := x:xs

  {- Note that popping an empty stack simply halts, with no effect -}
  def pop() =
    store? >h:t>
    store := t >>
    h

  {- A stack instance has no ongoing computation -}
  stop


{- Test the stack -}
val st = Stack()
st.push(3) >> st.push(5) >> st.pop() >> st.pop()

{-
OUTPUT:
3
-}
```

2.4.2. Example: Multicast

Here is a more complex example, which creates a multicast. Whenever a value is available on the source channel, it is read, and broadcasted to all current listeners. Listeners may be added with the addListener method.

```
def class Multicast(source) =
  val listeners = Ref([])

  def addListener(f) =
    listeners? >fs>
    listeners := f:fs

  {- The ongoing computation of a multicast -}
  repeat(source) >item> each(listeners?) >sink> sink(item)



{- Test the multicast -}

val c = Channel()
val mcast = Multicast(c.get)
val listenerA = Channel()
val listenerB = Channel()

{- At n seconds, broadcast n. Stop at 9 seconds. -}
  upto(10) >i> Rwait(1000*i) >> c.put(i) >> stop

{- Listener A joins at 1.5 seconds, hearing 2..9 -}
| Rwait(1500) >> mcast.addListener(listenerA.put) >> stop

{- Listener B joins at 6.5 seconds, hearing 7..9 -}
| Rwait(6500) >> mcast.addListener(listenerB.put) >> stop

{- Publish everything that Listener A hears -}
| repeat(listenerA.get) >a> ("A", a)

{- Publish everything that Listener B hears -}
| repeat(listenerB.get) >b> ("B", b)

{- Shortly after 10 seconds, close down the channels -}
| Rwait(10500) >>
    listenerA.close() >>
    listenerB.close() >>
    c.close() >>
    stop


{-
OUTPUT:PERMUTABLE
("A", 2)
("A", 3)
("A", 4)
("A", 5)
("A", 6)
("A", 7)
("B", 7)
("A", 8)
("B", 8)
("A", 9)
("B", 9)
-}
```

2.5. Datatypes

We have seen Orc's predefined data structures: tuples, lists, and records. Orc also provides the capability for programmers to define their own data structures, using a feature adopted from the ML/Haskell language family called *datatypes* (also called variants or tagged sums).

Datatypes are defined using the `type` declaration:

```
type Tree = Node(_,_,_) | Empty()
```

This declaration defines two new sites named `Node` and `Empty`. `Node` takes three arguments, and publishes a *tagged value* wrapping those arguments. `Empty` takes no arguments and does the same.

Once we have created these tagged values, we use a new pattern called a datatype pattern to match them and unwrap the arguments:

```
type Tree = Node(_,_,_) | Empty()
{- Build up a small binary tree -}
val l = Node(Empty(), 0, Empty())
val r = Node(Empty(), 2, Empty())
val t = Node(l,1,r)

{- And then match it to extract its contents -}
t >Node(l,j,r)>
l >Node(_,i,_)>
r >Node(_,k,_)>
( i | j | k )

{-
OUTPUT:PERMUTABLE
0
1
2
-}
```

One pair of datatypes is so commonly used that it is already predefined in the standard library: `Some(_)` and `None()`. These are used as return values for calls that need to distinguish between successfully returning a value (`Some(v)`), and successfully completing but having no meaningful value to return (`None()`). For example, a lookup function might return `Some(result)` if it found a result, or return `None()` if it successfully performed the lookup but found no suitable result.

2.6. Importing Resources

While the Orc language itself is expressive, and the Standard Library offers a number of useful sites and functions, it is often necessary to extend the capabilities of a program by writing new sites, using existing Java or Scala code, or making use of Orc code written by other programmers. Orc has three declarations, one corresponding to each of these three use cases.

2.6.1. Writing Custom Sites

The `import site` declaration allows a custom site to be imported and used in an Orc program. There are specific requirements that must be met by such sites; these are described in detail in the reference manual.

Suppose one had written a custom site, implemented in the Java class `my.example.site.ExampleSite`. The following code would make it available as a site named `Example` in an Orc program:

```
import site Example = "my.example.site.ExampleSite"
```

2.6.2. Using Java Classes

The `import class` declaration allows a Java class to be used as if it were an Orc site. The class constructor is imported as a site, and calls to that site return new Java objects whose methods and fields may be accessed using the dot notation. The specific details of this conversion are documented in the reference manual.

The following code imports and uses Java's `File`, `FileReader`, and `BufferedReader` classes to read the first line of a text file.

```
import class File = "java.io.File"
import class FileReader = "java.io.FileReader"
import class BufferedReader = "java.io.BufferedReader"
val f = File("example.txt")
val reader = BufferedReader(FileReader(f))
reader.readLine()
```

2.6.3. Including Source Files

The `include` declaration reads a text file containing Orc declarations and includes those declarations in the program as if they had occurred at the point where the `include` declaration occurred. Any declarations may be included: `val`, `def`, `import`, or even other `include` declarations. This provides a primitive form of modularity, where Orc code shared by many programs may be centralized in one or more include files.

An `include` declaration may name any URL, not just a local file. Thus, useful include files can be shared over the Internet directly.

```
{- Retrieve an include file from the Orc website and print the example message dec

include "http://orc.csres.utexas.edu/documentation/example.inc"
Println(example_message)
```

2.7. Type Checking

By default, Orc behaves as an untyped language. However, there is an optional static typechecker built into Orc, and an accompanying set of typed syntax and type annotations to support static typechecking. It can be enabled within the Orc Eclipse plugin, or via a command-line switch. The typechecker assures, up to the limitations of its algorithm, that no type errors will occur when the program is run. This is a useful sanity check on a program, making it easier to catch bugs early, or rule out certain causes for existing bugs. In some cases, it also speeds up program development.

A full description of the Orc typechecker is available in the Reference Manual. The reference manual section for each language feature describes how that feature interacts with the typechecker.

It is beyond the scope of this document to give a full tutorial on static type checking. However, we will briefly introduce the core concepts and describe the typical approach to using the type checker on an existing Orc program with no type information.

2.7.1. Type Inference

The Orc typechecker performs *type inference*, meaning that it can guess and confirm the type of many expressions without any extra information.

For example, the typechecker will happily check the following untyped program without any assistance:

```
val pi = 3.141592653
val radius = 7
val area = pi * radius * radius

Println("Area: " + area)
```

This program has type `Signal`, since the body expression is a `Println` call, which publishes a `signal`. The typechecker verifies that all operations in the program are type-correct.

If we had introduced a type error somewhere, the typechecker would catch it. For example, both of these programs fail to typecheck:

```
val pi = 3.141592653
val radius = 7
val area = "pi" * radius * radius  {- type error -}

Println("Area: " + area)
```

```
val pi = 3.141592653
val radius = 7
val area = pi * radius * radius

Println("Area ": area)   {- type error -}
```

2.7.2. Adding Type Information

When we begin adding function definitions to a program, the typechecker will need more information in order to operate correctly.

A defined function must have type information for each of its arguments. We add type information using the symbol `::`. This is similar to the requirements on methods in languages like Java or Scala:

```
{- Orc -}
def square(x :: Integer) = x * x

/* Scala */
def circleArea(x: int) = x * x

/* Java */
public int circleArea(int x) { return x * x; }
```

If the function is recursive, we must also give its return type:

```
def metronome() :: Signal = signal | Rwait(1000) >> metronome()
```

If the function has multiple clauses, or its arguments are complex patterns, this approach can be confusing. Instead of writing the types inline, we can write a *signature*, an extra declaration with the argument and return types:

```
def sum(List[Number]) :: Number  {-  a signature for 'sum' -}
def sum([]) = 0
def sum(h:t) = h + sum(t)
```

Notice the type of the list argument, `List[Number]`. `List` is a polymorphic (or "generic") type, which we will discuss further in the next section.

2.7.3. Polymorphism

The Orc type system has polymorphic, or "generic", types, such as `List`. These are the types of collections or containers whose contents might be of any type, so long as the type is consistent. For example, the list `[3,4,5]` has type `List[Integer]`, whereas `[[true], [false], []]` has type `List[Boolean]`.

A function may also be polymorphic, if it has polymorphic arguments. This must be made explicit in the function signature. For example, here is a list append function:

```
def append[X](List[X], List[X]) :: List[X]  {- Note the use of a type variable, X
def append([], l) = l
def append(h:t, l) = h:append(t,l)
```

The typechecker will allow `append` to be used on any two lists with the same type of elements; `X` is the name of that type. In a program that uses append, the typechecker is actually guessing a *type argument*, which gets bound to `X`.

When the programmer writes:

```
val a = [1,2,3]
val b = [4,5]
append(a,b)
```

the typechecker fills in a type argument for `append`:

```
val a = [1,2,3]
val b = [4,5]
append[Integer](a,b)
```

Sometimes the typechecker can't guess this type argument on its own. For example, the `Channel` site takes a type argument but no value arguments, so the typechecker doesn't have enough information available to guess the type argument when it encounters the call. It must be added explicitly:

```
{- Fails to typecheck -}
val c = Channel()
c.put(7)

{- Typechecks -}
val c = Channel[Integer]()
c.put(7)
```

This information may seem redundant, since `c.put(7)` obviously indicates that the channel will contain `Integer` values, but the typechecking algorithm does not use that information. It makes up for this limitation by providing more power in other areas.

This limitation is not unusual. Java constructors for generic classes have a similar requirement:

```
LinkedList<Integer> l = new LinkedList<Integer>();
```

Chapter 3. Syntactic and Stylistic Conventions

In this section we suggest some syntactic conventions for writing Orc programs. None of these conventions are required by the parser; newlines are used only to disambiguate certain corner cases in parsing, and other whitespace is ignored. However, following programming convention helps to improve the readability of programs, so that the programmer's intent is more readily apparent.

3.1. Parallel combinator

When the expressions to be combined are small, write them all on one line.

```
F | G | H
```

When the combined expressions are large enough to take up a full line, write one expression per line, with each subsequent expression aligned with the first and preceded by `|`. Indent the first expression to improve readability.

```
  long expression
| long expression
| long expression
```

A sequence of parallel expressions often form the left hand side of a sequential combinator. Since the sequential combinator has higher precedence, use parentheses to group the combined parallel expressions together.

```
( expression
| expression
) >x>
another expression
```

3.2. Sequential combinator

When the expressions to be combined are small, write a cascade of sequential combinators all on the same line.

```
F >x> G >y> H
```

When the expressions to be combined are individually long enough to take up a full line, write one expression per line; each line ends with the combinator which binds the publications produced by that line.

```
long expression  >x>
long expression  >y>
long expression
```

For very long expressions, or expressions that span multiple lines, write the combinators on separate lines, indented, between each expression.

```
very long expression
  >x>
very long expression
  >y>
very long expression
```

3.3. Pruning combinator

When the expressions to be combined are small, write them on the same line:

```
F <x< G
```

When multiple pruning combinators are used to bind multiple variables (especially when the scoped expression is long), start each line with a combinator, aligned and indented, and continue with the expression.

```
long expression
  <x< G
  <y< H
```

The pruning combinator is not often written in its explicit form in Orc programs. Instead, the `val` declaration is often more convenient, since it is semantically equivalent and mentions the variable `x` before its use in scope, rather than after.

```
val x = G
val y = H
long expression
```

Additionally, when the variable is used in only one place, and the expression is small, it is often easier to use a nested expression. For example,

```
val x = G
val y = H
M(x,y)
```

is equivalent to

```
M(G,H)
```

Sometimes, we use the pruning combinator simply for its capability to terminate expressions and get a single publication; binding a variable is irrelevant. This is a special case of nested expressions. We use the identity site `Let` to put the expression in the context of a function call.

For example,

```
x <x< F | G | H
```

is equivalent to

```
Let(F | G | H)
```

The translation uses a pruning combinator, but we don't need to write the combinator, name an irrelevant variable, or worry about precedence (since the expression is enclosed in parentheses as part of the call).

3.4. Declarations

When the body of a declaration spans multiple lines, start the body on a new line after the = symbol, and indent the entire body.

```
def f(x,y) =
    declaration
    declaration
    body expression
```

Apply this style recursively; if a def appears within a def, indent its contents even further.

```
def f(x,y) =
    declaration
    def helper(z) =
     declaration in helper
     declaration in helper
     body of helper
    declaration
    body expression
```

3.4.1. Ambiguous Declarations

The following situation could introduce syntactic ambiguity: the end of a declaration (def or val) is followed by an expression that starts with a non-alphanumeric symbol. Consider these example programs:

```
def f() =
  def g() = h
  (x,y)
```

```
def f() =
  val t = h
  (x,y)
```

```
def f() =
  val t = u
  -3
```

`(x,y)` may be interpreted as the parameter list of `h`, and `-3` as continuation of `u`, or they may be regarded as completely separate expressions (in this case, the goal expression of `def f`). To avoid this ambiguity, Orc imposes the following syntactic constraint:

An expression that follows a declaration begins with an alphanumeric symbol

To circumvent this restriction, if (x,y) is an expression that follows a declaration, write it as `signal >> (x,y)`. Similarly, write `signal >> -3`, in case `-3` is the goal expression in the above example. Note that there are many solutions to this problem; for example using `stop | (x,y)` is also valid.

Chapter 4. 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.

4.1. Channels

Orc has no communication primitives like pi-calculus channels[1] or Erlang mailboxes[2]. Instead, it makes use of sites to create channels of communication.

The most frequently used of these sites is `Channel`. 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.closeD()`, which returns a signal immediately.

[1]Robin Milner. 1999. *Communicating and Mobile Systems: The #-Calculus*. Cambridge University Press, New York, NY, USA.
[2]Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. 1996. *Concurrent programming in ERLANG (2nd ed.)*. Prentice Hall, Englewood Cliffs, NJ, USA.

4.2. Lists

In the introduction to the Orc language, we were introduced to lists: how to construct them, and how to match them against patterns. While it is certainly feasible to write a specific function with an appropriate pattern match every time we want to access a list, it is helpful to have a handful of common operations on lists and reuse them.

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)
```

Orc also 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 dispatches all of the mapped calls concurrently, and assembles the result list once they all return using a fork-join.

4.3. Streams

Sometimes a source of data is not explicitly represented by a list or other data structure. Instead, it is made available through a site, which returns the values one at a time, each time it is called. We call such a site a *stream*. It is analogous to an iterator in a language like Java. Functions can also be used as streams, though typically they will not be pure functions, and should only return one value. A call to a stream may halt, to indicate that the end of the data has been reached, and no more values will become available. It is often useful to detect the end of a stream using the otherwise combinator.

Streams are common enough in Orc programming that there is a library function to take all of the available publications from a stream; it is called `repeat`, and it is analogous to `each` for lists.

```
def repeat(f) = f() >x> (x | repeat(f))
```

The `repeat` function calls the site or function `f` with no arguments, publishes its return value, and recurses to query for more values. `repeat` should be used with sites or functions that block until a value is available. Notice that if any call to `f` halts, then `repeat(f)` consequently halts.

For example, it is very easy to treat a channel `c` as a stream, reading any values put on the channel as they become available:

```
repeat(c.get)
```

4.4. Mutable References

Variables in Orc are immutable. There is no assignment operator, and there is no way to change the value of a bound variable. However, it is often useful to have mutable state when writing certain algorithms. The Orc library contains two sites that offer simple mutable storage: `Ref` and `Cell`. It also provides the site `Array` to create mutable arrays.

A word of caution: References, cells, and other mutable objects may be accessed concurrently by many different parts of an Orc program, so race conditions may arise.

4.4.1. Rewritable references

The `Ref` site creates rewritable reference cells.

```
val r = Ref(0)
Println(r.read()) >>
r.write(2) >>
Println(r.read()) >>
stop

{-
OUTPUT:
0
2
-}
```

These are very similar to ML's `ref` cells. `r.write(v)` stores the value `v` in the reference `r`, overwriting any previous value, and publishes a signal. `r.read()` publishes the current value stored in `r`.

However, unlike in ML, a reference cell can be left initially empty by calling `Ref` with no arguments. A read operation on an empty cell blocks until the cell is written.

```
{- Create a cell, and wait 1 second before initializing it.
   The read operation blocks until the write occurs.
-}

val r = Ref()
r.read() | Rwait(1000) >> r.write(1) >> stop

{-
OUTPUT:
1
-}
```

4.4.2. Write-once references

The Orc library also offers write-once reference cells, using the `Cell` site. A write-once cell has no initial value. Read operations block until the cell has been written. A write operation succeeds only if the cell is empty; subsequent write operations simply halt.

```
{- Create a cell, try to write to it twice, and read it.
```

```
   The read will block until a write occurs
   and only one write will succeed.
-}

val r = Cell()
  Rwait(1000) >> r.write(2) >> Println("Wrote 2") >> stop
| Rwait(2000) >> r.write(3) >> Println("Wrote 3") >> stop
| r.read()

{-
OUTPUT:PERMUTABLE
2
Wrote 2
-}
```

Write-once cells are very useful for concurrent programming, and they are often safer than rewritable reference cells, since the value cannot be changed once it has been written. The use of write-once cells for concurrent programming is not a new idea; they have been studied extensively in the context of the Oz programming language [http://en.wikipedia.org/wiki/Oz_programming_language].

4.4.3. Syntax for manipulating references

Orc provides syntactic sugar for reading and writing mutable storage:

- `x?` is equivalent to `x.read()`. This operator is of equal precedence with the dot operator and function application, so you can write things like `x.y?.v?`. This operator is very similar to the C languages's `*` operator, but is postfix instead of prefix.
- `x := y` is equivalent to `x.write(y)`. This operator has higher precedence than the concurrency combinators and if/then/else, but lower precedence than any of the other operators.

Here is a previous example rewritten using this syntactic sugar:

```
{- Create a cell, try to write to it twice, and read it.
   The read will block until a write occurs
   and only one write will succeed.
-}

val r = Cell()
  Rwait(1000) >> r := 2 >> Println("Wrote 2") >> stop
| Rwait(2000) >> r := 3 >> Println("Wrote 3") >> stop
| r?

{-
OUTPUT:PERMUTABLE
2
Wrote 2
-}
```

4.4.4. Arrays

While lists are a very useful data structure, they are not mutable, and they are not indexed. However, these properties are often needed in practice, so the Orc standard library provides a function `Array` to create mutable arrays.

`Array(n)` creates an array of size `n` whose elements are all initially `null`. The array is used like a function; the call `A(i)` returns the `i`th element of the array `A`, which is then treated as a reference, just like the references created by `Ref`. A call with an out-of-bounds index halts, possibly reporting an error.

The following program creates an array of size 10, and initializes each index i with the ith power of 2. It then reads the array values at indices 3, 6, and 10. The read at index 10 halts because it is out of bounds (arrays are indexed from 0).

```
{- Create and initialize an array, then halt on out of bounds read -}
val a = Array(10)
def initialize(i) =
  if (i <: 10)
    then a(i) := 2 ** i  >>  initialize(i+1)
    else signal
initialize(0) >> (a(3)? | a(6)? | a(10)?)

{-
OUTPUT:PERMUTABLE
8
64
Error: java.lang.ArrayIndexOutOfBoundsException: 10
-}
```

The standard library also provides a helper function `fillArray` which makes array initialization easier. `fillArray(a, f)` initializes array `a` using function `f` by setting element `a(i)` to the first value published by `f(i)`. When the array is fully initialized, `fillArray` returns the array `a` that was passed (which makes it easier to simultaneously create and initialize an array). Here are a few examples:

```
{- Create an array of 10 elements; element i is the ith power of 2 -}
fillArray(Array(10), lambda(i) = 2 ** i) >a>
a(4)?

{-
OUTPUT:
16
-}
```

```
{- Create an array of 5 elements; each element is a newly created channel -}
fillArray(Array(5), lambda(_) = Channel())
```

```
{- Create an array of 2 channels -}
val A = fillArray(Array(2), lambda(_) = Channel())

{- Send true on channel 0,
   listen for a value on channel 0 and forward it to channel 1,
   and listen for a value on channel 1 and publish it.
-}

  A(0)?.put(true) >> stop
| A(0)?.get() >x> A(1)?.put(x) >> stop
| A(1)?.get()

{-
OUTPUT:
true
-}
```

Since arrays are accessed by index, there is a library function specifically designed to make programming with indices easier. The function `upto(n)` publishes all of the numbers from `0` to `n-1` simultaneously; thus, it is very easy to access all of the elements of an array simultaneously. Suppose we have an array `A` of `n` email addresses and would like to send the message `m` to each one.

```
upto(n) >i> A(i)? >address> Email(address, m)
```

4.5. Tables

Orc programs occasionally require a data structure that supports constant-time indexing but need not also provide mutable storage. For this purpose, an `Array` is too powerful. The standard library provides another structure called a `Table` to fill this role.

The call `Table(n,f)`, where `n` is a natural number and `f` a total function over natural numbers, creates and returns an immutable array of size `n` (indexed from 0), whose `i`th element is initialized to `f(i)`. A table can also be thought of as a partially memoized version of `f` restricted to the range [0, `n`). `Table` does not return a value until all calls to `f` have completed, and will halt if any call halts. Given a table `T`, the call `T(i)` returns the `i`th element of `T`. Notice that unlike array access, the `?` is not needed to subsequently dereference the return value, since it is not a mutable reference.

Tables are useful when writing algorithms which used a fixed mapping of indexes to some resources, such as a shared table of communication channels. Such a table could be constructed using the following code:

```
val size = 10
val channels = Table(size, lambda (_) = Channel())
```

Notice that the `lambda` ignores its argument, since each channel is identical. Here is another example, which memoizes the cubes of the first 30 natural numbers:

```
val cubes = Table(30, lambda(i) = i*i*i)
```

4.6. Loops

Orc does not have any explicit looping constructs. Most of the time, where a loop might be used in other languages, Orc programs use one of two strategies:

1. When the iterations of the loops can occur in parallel, write an expression that expands the data into a sequence of publications, and use a sequential operator to do something for each publication. This is the strategy that uses functions like `each`, `repeat`, and `upto`.

2. When the iterations of the loops must occur in sequence, write a tail recursive function that iterates over the data. Any loop can be rewritten as a tail recursion. Typically the data of interest is in a list, so one of the standard list functions, such as `foldl`, applies. The library also defines a function `while`, which handles many of the common use cases of while loops.

4.7. Parallel Matching

Matching a value against multiple patterns, as we have seen it so far, is a linear process, and requires a `def` whose clauses have patterns in their argument lists. Such a match is linear; each pattern is tried in order until one succeeds.

What if we want to match a value against multiple patterns in parallel, executing every clause that succeeds? Fortunately, this is very easy to do in Orc. Suppose we have an expression F which publishes pairs of integers, and we want to publish a signal for each 3 that occurs.

We could write:

```
F >(x, y)>
  ( Ift(x = 3) >> signal
  | Ift(y = 3) >> signal )
```

But there is a more general alternative:

```
F >x>
  ( x >(3, _)> signal
  | x >(_, 3)> signal )
```

The interesting case is the pair `(3,3)`, which is counted twice because both patterns match it in parallel.

This parallel matching technique is sometimes used as an alternative to pattern matching using function clauses, but only when the patterns are mutually exclusive.

For example,

```
def helper([]) = 0
def helper([_]) = 1
def helper(_:_:_) = 2
helper([4, 6])
```

is equivalent to

```
[4, 6] >x>
( x >[]> 0
| x >[_]> 1
| x >_:_:_> 2
)
```

whereas

```
def helper([]) = 0
def helper([_]) = 1
def helper(_) = 2
helper([5])
```

```
{-
OUTPUT:
1
-}
```

is *not* equivalent to

```
[5] >x>
( x >[]> 0
| x >[_]> 1
| x >_> 2
)

{-
OUTPUT:PERMUTABLE
1
2
-}
```

because the clauses are not mutually exclusive. Function clauses must attempt to match in linear order, whereas this expression matches all of the patterns in parallel. Here, it will match `[5]` two different ways, publishing both `1` and `2`.

4.8. Fork-join

One of the most common concurrent idioms is a *fork-join*: run two processes concurrently, and wait for a result from each one. This is very easy to express in Orc. Whenever we write a `val` declaration, the process computing that value runs in parallel with the rest of the program. So if we write two `val` declarations, and then form a tuple of their results, this performs a fork-join.

```
val x = F
val y = G
   signal >> (x,y)
```

Fork-joins are a fundamental part of all Orc programs, since they are created by all nested expression translations. In fact, the fork-join we wrote above could be expressed even more simply as just:

```
(F,G)
```

4.8.1. Example: Machine initialization

In Orc programs, we often use fork-join and recursion together to dispatch many tasks in parallel and wait for all of them to complete. Suppose that given a machine `m`, calling `m.init()` initializes `m` and then publishes a signal when initialization is complete. The function `initAll` initializes a list of machines.

```
def initAll([]) = signal
def initAll(m:ms) = ( m.init() , initAll(ms) ) >> signal
```

For each machine, we fork-join the initialization of that machine (`m.init()`) with the initialization of the remaining machines (`initAll(ms)`). Thus, all of the initializations proceed in parallel, and the function returns a signal only when every machine in the list has completed its initialization.

Note that if some machine fails to initialize, and does not return a signal, then the initialization procedure will never complete.

4.8.2. Example: Simple parallel auction

We can also use a recursive fork-join to obtain a value, rather than just signaling completion. 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 such an auction for a list of bidders (`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. Later we will see a different solution that addresses the issue of non-termination.

4.8.3. 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 )
```

4.9. 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.

4.10. Priority Poll

The otherwise combinator is also useful for trying alternatives in sequence. Consider an expression of the form `F`$_0$ `; F`$_1$ `; F`$_2$ `; ...`. 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 channels for available data. The list of channels is ordered by priority. The first channel in the list has the highest priority, so it is polled first. If it has no data, then the next channel is polled, and so on.

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

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

4.11. 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:

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

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 can publish `true` immediately regardless of whether the other variable is available. Therefore we run `Ift(a)  >>  true` and `Ift(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 `val` is necessary to select and publish only the first result.

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

```
Let( F | Rwait(1000) >> 0 )
```

4.12.1. Auction with timeout

In the auction example given previously, the auction may never complete if one of the bidders does not respond. We can add a timeout so that a bidder has at most 8 seconds to provide a bid:

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

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

4.12.2. 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) | (Rwait(t), false)
if b then G else H
```

Instead of using a boolean and conditional, we could use pattern matching:

```
val s = Some(F) | Rwait(t) >> None()
  s >Some(r)> G
| s >None()>  H
```

It is even possible to encapsulate timeout as a function.

```
def timeout(x, t) = Let(Some(x) | Rwait(t) >> None())
```

`timeout(F, t)` waits `t` milliseconds for F to publish a value. If F publishes `v` within the time limit, `timeout` returns `Some(v)`. Otherwise, it returns `None()` when the time limit is reached.

4.12.2.1. Timeout streams

We can also apply timeout to streams. Let's define a modified version of the `repeat` function as follows:

```
def repeatWithTimeout(f, t) =
  timeout(f(), t)
    >Some(x)>
  (x | repeatWithTimeout(f, t))
```

We call `f()` as before, but apply a timeout of `t` to the call. If a value becomes available from `f` before the timeout, then the call to `timeout` publishes `Some(x)`, which we match, and then publish `x` and recursively wait for further values from the stream.

However, if no value is available from `f` within the timeout, the call to `timeout` publishes `None()`. Since `None()` does not match the pattern, the entire expression halts, indicating that the end of the stream has been reached.

It is also possible to achieve this behavior with the existing `repeat` function, simply by changing the function passed to `repeat`:

```
def f'() = timeout(f(), t) >Some(x)> x
repeat(f')
```

4.13. 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( y | Rwait(1000) >> x )
```

4.14. 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 the previous chapter:

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

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"
| Rwait(500) >> metronome(1000) >> "tock"
```

4.15. 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.

4.15.1. 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.

4.15.1.1. 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 = Channel()
repeat(c.get) <<
    F >x> c.put(x) >> stop
  | Rwait(1000) >> c.closeD()
```

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, `Rwait(1000)` responds, triggering the call `c.closeD()`. The call `c.closeD()` 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.

4.15.1.2. Test Pruning

We can also decide to terminate based on the values published. This expression executes F until it publishes a negative number, and then terminates it:

```
val c = Channel()
repeat(c.get) <<
  F >x>
    (if x >= 0
        then c.put(x) >> stop
        else c.closeD())
```

Each value published by F is tested. If it is non-negative, it is placed on channel `c` (silently) and read by `repeat(c.get)`. If it is negative, the channel is closed, publishing a signal and causing the termination of F.

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

4.15.1.4. Publication Limit

We can limit an expression to *n* publications, rather than just one. Here is an expression which executes F until it publishes 5 values, and then terminates it.

```
val c = Channel()
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.closeD()
```

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`.

4.15.2. 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 = Cell()
signal >>
   (F <x< c.read()) | (G >x> c.write(x))
```

4.15.3. Sequencing 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 = Channel()
repeat(c.get) | (F >x> c.put(x) >> stop ; c.close() >> G)
```

We use `c.close()` instead of the more common `c.closeD()` 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.

4.16. 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. However, if `f` is associative, we can overcome this restriction and implement fold concurrently. If `f` is also commutative, we can further increase concurrency.

4.16.1. Associative Fold

We first consider the case when the reduction operator is associative. We define `afold(f,xs)` where `f` 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.

```
def afold(_, [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. Thus, the call `f(x,y)` executes in parallel with the recursive call `step(xs)`. As a result, all calls to `f` execute concurrently within each iteration of `afold`.

4.16.2. 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 a associative and commutative binary function and `xs` is a non-empty list. The implementation initially copies all list items into a channel 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 channel, reduces them, and places the result back in the channel. Each pair of items is reduced in parallel as they become available. The last item in the channel is the result of the overall fold.

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

  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)
```

Chapter 5. Larger Examples

In this section we show a few larger Orc programs to demonstrate programming techniques. There are many more such examples available at the Orc Web site, on the community wiki [http://orc.csres.utexas.edu/wiki/Wiki.jsp?page=WikiLab].

5.1. 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 [1], 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.

```
{- Dining Philosophers -}

{- Randomly swap order of fork pick-up -}
def shuffle(a,b) = if (Random(2) = 1) then (a,b) else (b,a)

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

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

{- Define a philosopher -}
def phil(n,a,b) =
  def thinking() =
    Println(n + " thinking") >>
    (if (Random(10) <: 9)
      then Rwait(Random(1000))
      else stop)
  def hungry() = take((a,b))
  def eating() =
    Println(n + " eating") >>
    Rwait(Random(1000)) >>
    Println(n + " done eating") >>
    drop(a,b)
  thinking() >> hungry() >> eating() >> phil(n,a,b)

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

{- Test the definitions -}
val fork = Semaphore(1)
philosophers(5,fork,fork)

{-
```

[1] Daniel Lehmann and Michael O. Rabin. 1981. On the advantages of free choice: a symmetric and fully distributed solution to the dining philosophers problem. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL '81). ACM, New York, NY, USA, 133-138.

```
OUTPUT:EXAMPLE
5 thinking
4 thinking
3 thinking
2 thinking
1 thinking
1 eating
4 eating
...
-}
```

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(a,b)` 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.acquireD()` 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 call `philosophers(n,a,b)` 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.

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.

5.2. Hygienic Dining Philosophers

Here we implement a different solution to the Dining Philosophers problem, described in "The Drinking Philosophers Problem", by K. M. Chandy and J. Misra [2]. Briefly, this algorithm efficiently and fairly solves the dining philosophers problem for philosophers connected in an arbitrary graph (as opposed to a simple ring). The algorithm works by augmenting each fork with a clean/dirty state. Initially, all forks are dirty. A philosopher is only obliged to relinquish a fork to its neighbor if the fork is dirty. On receiving a fork, the philosopher cleans it. On eating, the philosopher dirties all forks. For full details of the algorithm, consult the original paper.

```
{- The "hygenic solution to the diners problem",
   described in "The Drinking Philosophers Problem", by
   K. M. Chandy and J. Misra.
-}

{-
Use a Scala set implementation.
Operations on this set are not synchronized.
-}
import class ScalaSet = "scala.collection.mutable.HashSet"

{-
Make a set initialized to contain
the items in the given list.
-}
def Set(items) = ScalaSet() >s> joinMap(s.add, items) >> s


{-
Start a philosopher process; never publishes.

name: identify this process in status messages
mbox: our mailbox
missing: set of neighboring philosophers holding our fork
-}
def philosopher(name, mbox, missing) =
  val send = mbox.put
  val receive = mbox.get
  -- deferred requests for forks
  val deferred = Channel()
  -- forks we hold which are clean
  val clean = Set([])

  def sendFork(p) =
    missing.add(p) >>
    p(("fork", send))

  def requestFork(p) =
    clean.add(p) >>
    p(("request", send))
```

[2] K. Mani Chandy and Jayadev Misra. 1984. The drinking philosophers problem. *ACM Trans. Program. Lang. Syst.* 6, 4 (October 1984), 632-646.

```
  -- While thinking, start a timer which
  -- will tell us when we're hungry
  def digesting() =
      Println(name + " thinking") >>
      thinking()
    | Rwait(Random(30)) >>
      send(("rumble", send)) >>
      stop

  def thinking() =
    def on(("rumble", _)) =
      Println(name + " hungry") >>
      map(requestFork, missing.toList()) >>
      hungry()
    def on(("request", p)) =
      sendFork(p) >>
      thinking()
    on(receive())

  def hungry() =
    def on(("fork", p)) =
      missing.remove(p) >>
      (
        if missing.isEmpty() then
          Println(name + " eating") >>
          eating()
        else hungry()
      )
    def on(("request", p)) =
      if clean.contains(p) then
        deferred.put(p) >>
        hungry()
      else
        sendFork(p) >>
        requestFork(p) >>
        hungry()
    on(receive())

  def eating() =
    clean.clear() >>
    Rwait(Random(10)) >>
    map(sendFork, deferred.getAll()) >>
    digesting()

  digesting()

{-
Create an NxN 4-connected grid of philosophers.  Each philosopher holds the
fork for the connections below and to the right (so the top left philosopher
holds both its forks).
-}
def philosophers(n) =
  {- channels -}
  val cs = uncurry(Table(n, lambda (_) = Table(n, ignore(Channel))))

  {- first row -}
  philosopher((0,0), cs(0,0), Set([]))
  | for(1, n) >j>
    philosopher((0,j), cs(0,j), Set([cs(0,j-1).put]))

  {- remaining rows -}
  | for(1, n) >i> (
      philosopher((i,0), cs(i,0), Set([cs(i-1,0).put]))
      | for(1, n) >j>
        philosopher((i,j), cs(i,j), Set([cs(i-1,j).put, cs(i,j-1).put]))
    )

philosophers(2)

{-
OUTPUT:EXAMPLE
(0, 0) thinking
(0, 1) thinking
(1, 0) thinking
(1, 1) thinking
(1, 0) hungry
(0, 0) hungry
(0, 1) hungry
(1, 1) hungry
(1, 1) eating
(1, 1) thinking
...
-}
```

Our implementation is based on the actor model [http://en.wikipedia.org/wiki/Actor_model] of concurrency. An actor is a state machine which reacts to messages. On receiving a message, an actor can send asynchronous messages to other actors, change its state, or create new actors. Each actor is single-threaded and processes messages sequentially, which makes some concurrent programs easier to reason about and avoids explicit locking. Erlang [http://www.erlang.org/] is one popular language based on the actor model.

Orc emulates the actor model very naturally. In Orc, an actor is an Orc thread of execution, together with a `Channel` which serves as a mailbox. To send a message to an actor, you place it in the actor's mailbox, and to receive a message, the actor gets the next item from the mailbox. The internal states of the actor are represented by functions: while an actor's thread of execution is evaluating a function, it is considered to be in the corresponding state. Because Orc implements tail-call optimization [http://en.wikipedia.org/wiki/Tail_call], state transitions can be encoded as function calls without running out of stack space.

In this program, a philosopher is implemented by an actor with three primary states: `eating`, `thinking`, and `hungry`. An additional transient state, `digesting`, is used to start a timer which will trigger the state change from `thinking` to `hungry`. Each state is implemented by a function which reads a message from the mailbox, selects the appropriate action using pattern matching, performs the action, and finally transitions to the next state (possibly the same as the current state) by calling the corresponding function.

Forks are never represented explicitly. Instead each philosopher identifies a fork with the "address" (sending end of a mailbox) of the neighbor who shares the fork. Every message sent includes the sender's address. Therefore when a philosopher receives a request for a fork, it knows who requested it and therefore which fork to relinquish. Likewise when a philosopher receives a fork, it knows who sent it and therefore which fork was received.

5.3. Readers-Writers

Here we present an Orc solution to the readers-writers problem [http://en.wikipedia.org/wiki/Readers-writers_problem]. Briefly, the readers-writers problem involves concurrent access to a mutable resource. Multiple readers can access the resource concurrently, but writers must have exclusive access. When readers and writers conflict, different solutions may resolve the conflict in favor of one or the other, or fairly. In the following solution, when a writer tries to acquire the lock, current readers are allowed to finish but new readers are postponed until after the writer finishes. Lock requests are granted in the order received, guaranteeing fairness. Normally, such a service would be provided to Orc programs by a site, but it is educational to see how it can be implemented directly in Orc.

```
{- A solution to the readers-writers problem -}

{- Queue of lock requests -}
val m = Channel()
{- Count of active readers/writers -}
val c = Counter()

{- Process requests in sequence -}
def process() =
  {- Grant read request -}
  def grant((false,s)) = c.inc() >> s.release()
  {- Grant write request -}
  def grant((true,s)) =
    c.onZero() >> c.inc() >> s.release() >> c.onZero()
  {- Goal expression of process() -}
  m.get() >r> grant(r) >> process()

{- Acquire the lock: argument is "true" if writing -}
def acquire(write) =
  val s = Semaphore(0)
  m.put((write, s)) >> s.acquire()

{- Release the lock -}
def release() = c.dec()

-------------------------------------------------

{- These definitions are for testing only -}
def reader(start) = Rwait(start) >>
  acquire(false) >> Println("START READ") >>
  Rwait(1000) >> Println("END READ") >>
  release() >> stop
def writer(start) = Rwait(start) >>
  acquire(true) >> Println("START WRITE") >>
  Rwait(1000) >> Println("END WRITE") >>
  release() >> stop

Let(
    process()  {- Output:     -}
  | reader(10) {- START READ  -}
  | reader(20) {- START READ  -}
```

```
               {- END READ    -}
               {- END READ    -}
  | writer(30) {- START WRITE -}
               {- END WRITE   -}
  | reader(40) {- START READ  -}
  | reader(50) {- START READ  -}
               {- END READ    -}
               {- END READ    -}
  {- halt after the last reader finishes -}
  | Rwait(60) >> acquire(true)
)

{-
OUTPUT:EXAMPLE
END READ
START WRITE
END WRITE
START READ
START READ
END READ
END READ
signal
-}
```

The lock receives requests over the channel `m` and processes them sequentially with the function `grant`. Each request includes a boolean flag which is true for write requests and false for read requests, and a `Semaphore` which the requester blocks on. The lock grants access by releasing the semaphore, unblocking the requester.

The counter `c` tracks the number of readers or writers currently holding the lock. Whenever the lock is granted, `grant` increments `c`, and when the lock is released, `c` is decremented. To ensure that a writer has exclusive access, `grant` waits for the `c` to become zero before granting the lock to the writer, and then waits for `c` to become zero again before granting any more requests.

5.4. 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 `t` by calling `part` to partition the subarray and then recursively sorting the partitions.

```
{- Perform Quicksort on a list -}

def quicksort(a) =

  def swap(x, y) = a(x)? >z> a(x) := a(y)? >> a(y) := z

  {- Partition the elements based on pivot point 'p' -}
  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

    signal >>
      (lr(s), rl(t)) >(s', t')>
      ( Ift (s' + 1 <: t') >> swap(s', t') >> part(p, s'+1, t'-1)
      | Ift (s' + 1 = t') >> swap(s', t') >> s'
      | Ift (s' + 1 :> t') >> t'
      )

  {- Sort the elements -}
  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)

val a = Array(3)
a(0) := 1 >>
a(1) := 3 >>
a(2) := 2 >>

quicksort(a) >> arrayToList(a)

{-
OUTPUT:
[1, 2, 3]
```

[3]C. A. R. Hoare. 1961. Algorithm 63: Partition, Algorithm 64: Quicksort, and Algorithm 65: Find. *Commun. ACM* 4, 7 (July 1961), 321-322.

```
-}
```

The function `part` partitions the subarray given by indices `s` through `t` into two partitions, one containing values less than or equal to `p` and the other containing values > `p`. The last index of the lower partition is returned. The value at `a(s-1)` is assumed to be less than or equal to `p` --- this is satisfied by choosing `p = a(s-1)?` 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.

5.5. 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*, 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.

```
{- This program requires the Orchard environment to run -}
include "forms.inc"
include "mail.inc"

val during = Interval(LocalDate(2009, 9, 10),
                      LocalDate(2009, 10, 17))
val invitees = ["john@example.com", "jane@example.com"]

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 Web site [http://orc.csres.utexas.edu/tryorc.shtml].