A User's Guide to Orc

David Kitchin

March 17, 2008

Contents

1 Introduction

Orc is an orchestration language, which expresses structured concurrency. The implementation of Orc extends the core calculus with syntactic support for many of the conveniences of modern programming languages.

2 Basic Expressions

2.1 Constants

The simplest program we can write is a constant value, for example:

```
3
```

This publishes the value 3 immediately.

Orc supports integer constants (... `-1`, `0`, `1` ...), booleans (`true` and `false`), strings (`"orc"`), and a special *signal* value, written `()`.

2.2 Variables

A variable name on its own is also a valid Orc program:

```
y
```

This waits for `y` to be bound, then publishes the value bound to `y`. If `y` is already bound, it will publish immediately. It may never publish if `y` remains unbound forever.

2.3 Sites

When an identifier is followed by a parenthesized sequence of arguments, it is a site call.

```
M(x,1)
```

In this case the site `M` is being called with the arguments `x` and `1`. This program will wait for `x` to become bound, and then invoke the service associated with the name `M`, giving it `x`'s value and the value 1 as arguments.

A site may be invoked with no arguments, in which case we write an empty sequence `()`:

```
N()
```

Note that sites are values, so site identifiers are just variable names, and obey all of the same rules as ordinary variables. For example, the variable `N` in the program above could be unbound, in which case the call must wait for `N` to become bound to know what service to invoke.

Also note that any expression can be used as an argument, so for example the following program is valid:

```
M(N(1),N(2))
```

Here, the service bound to `N` is invoked twice, with two different arguments. These two invocations occur in parallel. When both invocations have completed and published values, the service bound to `M` is invoked with those values as arguments.

2.4 Operators

Orc supports a standard set of integer arithmetic, logical, and comparative operators. These operators are written in infix notation.

Arithmetic		Comparison		Logical	
`+`	addition	`=`	equal	`&&`	logical and
`-`	subtraction	`/=`	unequal	`||`	logical or
`*`	multiplication	`<:`	less than	`~`	negation
`/`	integer division	`:>`	greater than		
`%`	modulus	`<=`	less than or equal		
		`>=`	greater than or equal		

The operators have the standard C-like precedence and associativity. And, as with site calls, they can be nested:

```
2+3 :> 1*4  &&  true /= ~true
```

2.5 `null`

The special expression `null` is the silent expression; it never publishes anything.

2.6 Projection

Sometimes values have an object-like structure, and contain named members, which could be thought of as methods or fields. These are accessed using the `.` syntax:

```
x.foo
```

The value bound to `x` is queried to see if it has a member labeled 'foo', and if so, it publishes that member. This syntax is commonly used to access services as if they were methods of an object:

```
c.put(4)
```

This program queries the value `c` for the member 'put', which is a service. That service is published, and then invoked with the argument `4`.

3 Combinators

3.1 Bar

The bar combinator `|` works exactly like its counterpart in the Orc calculus. It executes two programs in parallel:

```
M(0) | N(1,2)
```

The bar combinator is fully associative and commutative.

3.2 Push

The push combinator `>x>` is just like the sequential composition of the Orc calculus. It executes its left side, then for each publication produced by the left side, it starts a new copy of the right side with `x` bound to that value:

```
(1 | 2) >y> N(y+1)
```

When the bound variable is not needed, push can be written as `>>`, with no variable name:

```
M(0) >> M(1) >> M(2)
```

The push combinator is right-associative. It has higher precedence than bar.

3.3 Pull

The pull combinator `<x<` is just like the asymmetric composition, or 'where' combinator, of the Orc calculus. It executes its left and right sides in parallel. Expressions using `x` on the left side block until it is bound. The first publication of the right side is bound to `x` and causes the rest of the right side to terminate.

```
x+1 <x< ( M(0) | M(1) )
```

When the bound variable is not needed (though this is rare), pull can be written as `<<`, with no variable name:

```
N() << ( M(0) | M(1) )
```

The pull combinator is left-associative. It has lower precedence than bar.

4 Comments

The implementation supports two kinds of comments:

```
-- Single line comments must begin with two dashes(--)
   -- They may only be preceded by whitespace.

   {-
     Multiline comments begin with {-
     and end with -}.
   -}

   {- They can be {- nested -} -}

M()
  {- They may appear anywhere, -} >x>
  {- even in the middle of an expression. -} N(x)
```

5 Data Structures

While services may make their own data types available to Orc, there are a few primitive data types that are important enough to deserve explicit language support.

5.1 Tuples

A parenthesized sequence of expressions constructs a tuple of values:

```
(1,x,2)
```

This publishes the tuple (1,v,2), where v is the value bound to `x`. Like site calls, a tuple will block until all of its arguments have published a value. Furthermore, expressions can be nested in tuples.

Note that it is not possible to construct a tuple with less than two items. `()` is interpreted as the signal value. `(f)` is interpreted as the parenthesization of the expression `f`, not as a unary tuple.

5.2 Lists

Lists of values can be constructed in two ways. First, the list can be defined explicitly:

```
[1, 2, x, 4]
```

This is similar to tuple creation.

There is also a cons operator, `:`, which publishes a new list resulting from the cons of an element onto an existing list:

```
[2,3] >x> 1:x
```

This publishes the list `[1,2,3]`.

The empty list is written `[]`.

6 Pattern Matching

In addition to building data structures, we also need to be able to inspect them. The Orc implementation extends the core calculus with *patterns*, which replace variables in the push and pull combinators, allowing a publication to bind multiple variables to different components of a complex value.

In addition, a pattern can also *refuse* a value which does not correctly match the pattern's structure. When a pattern refuses a value in a push `>p>`, then no new copy of the right hand side is executed. When a pattern refuses a value in a pull `<p<`, then the right hand side is not terminated; it continues as if no publication had occurred. A pattern which may refuse some value is called *refutable*; if it cannot refuse any value then it is *irrefutable*.

6.1 Variables

Variables are the most basic patterns. A variable is irrefutable and simply binds the value to that variable. Patterns must be *linear*, meaning that a pattern may not mention the same variable more than once.

6.2 Wildcard

A wildcard pattern `_` matches any value, and does not bind any variables. It is the same as binding a variable which is not used anywhere. In fact the empty push (`>>`) and empty pull (`<<`) are just shorthand for `>_>` and `<_<` respectively. A wildcard is irrefutable.

6.3 Literals

A literal pattern can be any of the constant values: an integer, boolean, string, or signal. It will only match that value, and will refuse any others.

6.4 Tuples

A tuple of patterns matches a tuple of the same length, and additionally matches each member of the tuple against each of its member patterns. For example, the expression

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

will publish `2` and `3` but not `4`. The pattern `(1,x)` refuses the value `(5,4)`.

6.5 Lists

A list of patterns matches a list of the same length, and additionally matches each member of the list against each of its member patterns. For example, the expression

```
( [1] | [2,3] | [4,5,6] )  >[x,y]>  x+y
```

publishes only `5`; the pattern refuses the other two lists because they are of the wrong length.

6.6 Cons

It is also possible to use the cons `:` operator in a pattern, to split a list into its head and tail:

```
[1,2,3]  >h:t>  ( h | t )
```

This publishes `1` and `[2,3]`. Note that a cons pattern will refuse the empty list.

6.7 Bang

A bang pattern, written `!p`, will publish the value that matches the pattern if the match is successful:

```
(1,2,3)  >(!x,y,z)>  (z,z)
```

This publishes `1` and `(3,3)`.

Note that a bang pattern will not publish if the overall match fails, even if its particular match succeeds:

```
( (1,2,3) | (4,5,6) )  >(1,!x,y)>  null
```

This publishes only `2`. Even though the pattern `x` matches the value `5`, the overall pattern `(1,x,y)` refuses the value `(4,5,6)`, so `5` is not published.

6.8 As

The keyword `as` can be used to capture a whole subpattern and bind it to a variable:

```
(1,(2,3))  >(x,(2,z) as w)>  w
```

This publishes `(2,3)`.

7 Defining Expressions

7.1 Simple definitions

Simple expression definitions in the implementation are very similar to their counterparts in the Orc core calculus. We use the keyword `def` instead of the symbol $\underline{\Delta}$.

```
def E(x,y) = x | y | x+y
E(2,3)
```

This defines the expression `E` and then calls it, publishing `2`, `3`, and `5`. Note that a definition is always defined in some scope; in this case the scope is the expression `E(2,3)`.

We might have instead written:

```
(
 def E(x,y) = x | y | x+y
 E(2,3)
)
 | E(4,5)
```

But this will generate an error: the definition of `E` is not in scope for the call `E(4,5)`. Note that this is a departure from the core calculus, which assumes that all definitions are available at the top level.

Defined expressions have a call-by-name semantics, just like in the core calculus. Consider the following program:

```
def E(x,y) = x | y | x+y
E(a,b)
    <a< null
    <b< 4
```

This will publish `4`, even though `a` is never bound. The call `E(a,b)` occurs immediately, effectively executing the expression `a | b | a+b` in its place.

7.2 Recursion

Definitions can be recursive; that is, the name of a definition is bound in its own body.

```
def countdown(n) = if(n :> 0) >> ( n | countdown(n-1) )
countdown(3)
```

This publishes `3`, `2`, `1`.

7.3 Mutual recursion

Mutual recursion is also supported:

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

There is no special keyword for mutual recursion; any contiguous sequence of definitions is assumed to be mutually recursive.

7.4 Closures

Definitions are actually values, just like any other value. Defining an expression creates a special value called a *closure* and binds it. Thus, it can be published, or put into a data structure, like any other value:

```
f(1) | f(3)
<f<
(def E(x) = x+1
 E)
```

This publishes `2` and `4`. It defines the expression `E`, then publishes that definition as a closure, which is then bound to the variable `f` and called twice.

7.5 Anonymous definitions

Sometimes one would like to create a closure directly without bothering to name the defined expression. There is a special keyword `lambda` for this purpose:

```
lambda (x) = x+1
{-    equivalent to (def E(x) = x+1  E)    -}
```

7.6 Patterns

A defined expression, just like the push and pull combinators, may have patterns instead of variables as its arguments:

```
def E([x,y]) = x | y
```

This definition publishes two elements of a list, but only if its argument is in fact a two-element list. Otherwise, the call remains silent.

Note that using patterns as arguments interacts with the call-by-name semantics of defined expressions: if a pattern is refutable, then the body of the expression cannot execute until that argument is bound.

7.7 Clauses

The combination of defined expressions and pattern matching offers a much more powerful capability: *clausal* definition of expressions. 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.

Any contiguous sequence of definitions with the same name and different arguments is interpreted as a clausal definition, where each individual definition is a clause of the larger definition. When the definition 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.

Mutual recursion and clausal definitions are allowed to occur together. For example, this definition takes a list and publishes a new list with every other element repeated:

```
def stutter([]) = []
def stutter(h:t) = h:h:mutter(t)
def mutter([]) = []
def mutter(h:t) = h:stutter(t)
```

8 Declarations

A declaration is a statement that modifies the environment of some expression (usually by binding a variable), but does not publish any values on its own. A declaration always has a scope.

We have already seen `def`, which is a declaration, but there are others.

8.1 val

A `val` declaration binds the first publication of an expression to a variable. It works the same way as a pull, and is just a convenient shorthand for writing a pull in prefix form:

```
val x = 4+5
M(x)
{-   same as M(x) <x< 4+5   -}
```

8.2 site

A `site` declaration instantiates a Java object to be used as a site in an Orc program:

```
{-  Define the Buffer site  -}
site Buffer = orc.lib.state.Buffer
```

8.3 class

A `class` declaration looks very similar to a `site` declaration, but serves a different purpose. Rather than creating a Java object to be used as a site, this declaration actually creates a proxy for the class's constructor, which can then be invoked, and its publications used like Java objects by using the dot syntax:

```
class Str = java.lang.String
Str("foo") >s> s.concat("bar")
```

8.4 include

An include declaration names a file from which a sequence of other declarations are loaded, as if they had been substituted directly into the program at that point. This is a convenient way to organize large groups of declarations.

```
{-  include the standard list functions  -}
include "inc/orc/list.inc"
```

9 Time

One of Orc's strongest features is its inclusion of time as a fundamental capability. The Orc implementation supports two forms of timing: real time (using the system clock via Java) and simulated time (using a logical clock).

9.1 Real Time

The site `Rtimer` allows synchronization based on real time. A call `Rtimer(d)` waits for `d` milliseconds, and then publishes a signal.

The expression `Metronome`, which publishes a signal every second, is defined using `Rtimer`:

```
def Metronome() = () | Rtimer(1000) >> Metronome()
```

9.2 Logical Time

The site `Ltimer` behaves like `Rtimer`, except that it is based on *simulated* time. A call `Ltimer(n)` waits for `n` units of simulated time, and then publishes the current value of the logical clock. The implementation guarantees that logical timer calls will publish responses, one at a time, in the correct order.

For example, the iteration:

```
each([]) = null
each(h:t) = h | Ltimer(1) >> t
each([1,2,3,4])
```

will *always* publish `1`, `2`, `3`, `4`, in that order.